深入分析Linux内核源代码

陈莉君　编著

人 民 邮 电 出 版 社

图书在版编目（CIP）数据
深入分析Linux内核源代码/陈莉君编著．—北京：人民邮电出版社，2002.8

ISBN 7-115-10525-1

I. 深… II. 陈… III. Linux操作系统 IV. TP316.89

中国版本图书馆CIP数据核字（2002）第056978号

内 容 提 要

自由软件Linux操作系统源代码的开放，为我们掌握操作系统核心技术提供了良好的条件。本书共分13章，对Linux内核2.4版的源代码进行了较全面的分析，既包括对中断机制、进程调度、内存管理、进程间通信、虚拟文件系统、设备驱动程序及网络子系统的分析，也包括对Linux整体结构的把握、Linux的启动过程的分析及Linux独具特色的模块机制的分析与应用等。其中重点剖析了Linux内核中最基础的部分：进程管理、内存管理及文件管理。

本书对于那些准备进入Linux操作系统内部，阅读Linux内核源代码以及在内核级进行程序开发的读者具有非常高的参考价值。同时，操作系统实现者、系统程序员、Linux应用开发人员、嵌入式系统开发人员、系统管理员、在校的大学生和研究生及对Linux感兴趣的用户均可在阅读本书中受益。

深入分析Linux内核源代码

◆ 编　　著　陈莉君
　责任编辑　魏雪萍

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　　电子函件　315@ ptpress.com.cn
　网址　http://www.ptpress.com.cn
　读者热线：010-67180876
　北京汉魂图文设计有限公司制作
　北京印刷厂印刷
　新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
　印张：
　字数：　千字　　　　2002年8月第1版
　印数：1-　000册　　　2002年8月北京第1次印刷

ISBN 7-115-10525-1/TP・3021

定价：00.00元

本书如有印装质量问题，请与本社联系 电话：(010)67129223

前　　言

如果说 Linux 的出现是一个偶然，那么，席卷全球的 Linux 热潮则是一个奇迹，Linux 正以势不可挡的趋势迅猛发展，其发展前景是无法预测的。

有人说，“Linux 不就是类 UNIX 吗？”是的，它的外在表现形式确实与 UNIX 完全兼容，这也是它赖以生存的基本条件。但是，它的内涵则完全不同，这首先体现在其源代码全部重写及开放，其次是它的快速更新和发展，而更重要的是世界范围内众多计算机爱好者能通过 Internet 参与开发，由此可见，借助于 Internet 的肥沃土壤，Linux 的迅速发展是毫无置疑的！

实际上，Linux 最本质的东西体现在其“自由”和“开放”的思想，“自由”意味着世界范围内的知识共享，而“开放”则意味着 Linux 对所有的人都敞开大门，在这开放而自由的天地里，你的创造激情可以得到充分的发挥。

Linux 内核源代码的开放给希望深入操作系统内部世界的人敞开无私的胸怀，我们有幸走进了这个世界，这是一个神奇、错综复杂而又充满诱惑的世界，让喜欢迎接挑战的人们可以充分检验自己的勇气和耐力。

Linux 内核全部源代码是一个庞大的世界，大约有 200 多万行，占 60MB 左右的空间。因此，如何在这庞大而复杂的世界中抓住主要内容，如何找到进入 Linux 内部的突破口，又如何能把 Linux 的源代码变为自己的需要，这就是本书要探讨的内容。

首先，本书的第一章领你走入 Linux 的大门，让你对 Linux 内核的结构有一个整体的了解。然后，第二章介绍了分析 Linux 源代码应具备的基本硬件知识，这是继续向 Linux 内核迈进的必备条件。中断作为操作系统中发生最频繁的一个活动，本书用一章的内容详细描述了中断在操作系统中的具体实现机制。

众所周知，操作系统中最核心的内容就是进程管理、内存管理和文件管理。本书用大量的篇幅描述了这三部分内容，尤其对最复杂的虚拟内存管理进行了详细的分析，其中对内存初始化部分的详细描述将对嵌入式系统的开发者有所帮助。

在对 Linux 内核有一定了解后，读者可能希望能够利用内核函数进行内核级程序的开发，例如开发一个设备驱动程序。Linux 的模块机制就是支持一般用户进行内核级编程。另外，读者在进行内核级编程时还可以快速查阅本书附录部分提供的 Linux 内核 API 函数。

网络也是 Linux 中最复杂的部分之一，这部分内容足可以写一本书。本书仅以面向对象的思想为核心，分别对网络部分中的四个主要对象：协议、套接字、套接字缓冲区及网络设备接口进行了分析。有了对这四个对象的分析，再结合文件系统、设备驱动程序的内容，读者就可以具体分析自己感兴趣的相关内容。

Linux 在不断地发展，本书介绍的版本为 Linux 2.4.16。尽管本书力图反映 Linux 内核较本质的东西，但由于笔者的知识有限，对有些问题的理解难免有偏差，甚至可能有“Bug”，希望读者能尽可能多地发现它，以共同对本书进行改进和完善。

在本书的编写的过程中，笔者查阅了大量的资料，也阅读了大量的源代码，但本书中反映的内容也仅仅是 Linux 的主要内容。因为一本书的组织形成是一种线性结构，而知识本身的组织结构是一种树型结构，甚至是多线索的网状结构，因此，在本书的编写过程中，笔者深感书的表现能力非常有限，一本书根本无法囊括全部。在参考书目中，我们将给出主要的参考书及主要网站的相关内容。

本书的第一版是《Linux 操作系统内核分析》，在第一版的编写过程中，康华、季进宝、陈轶飞、张波、张蕾及胡清俊等参与了编写。第一版出版后得到了很多读者的充分肯定和赞扬，并授权台湾地区出版。在本次改版的过程中，依然保留了第一版的风格，但加深了对进程管理、内存管理及文件管理等众多内容的剖析。

这次改版由于时间仓促，加之作者的水平有限，书中有些术语的表达可能不妥，有些内容的分析也可能不够准确，敬请读者朋友批评指正。我的联系方式是：cljun@xiyou.edu.cn。

编者

2002 年 7 月

目　　录

第一章　走进 Linux

对经常使用计算机的人来说 ，时常会感到操作系统是一个神奇、神秘而又几乎无所不能的“ 上帝 ”。一打开计算机，我们首先看到的是操作系统，所有软件的运行都离不开它，它给我们带来一个个的惊喜，但有时也带来烦恼和不安。

实际上，很多人都有这样强烈的愿望，即“ 上帝 ”到底是怎样操纵这一切的？UNIX 操作系统曾经敞开 UNIX 操作系统的胸怀，让我们窥视到它的内在机制，但它毕竟属于“ 贵族 ”阶层，我们大多数人并不能使用上它。

Windows 以平民的身份来到我们中间，我们欢呼它的友好和平易近人，正因为 Windows，才使得计算机走进我们寻常百姓家，使得计算机普及成为现实。但 Windows 有时也“ 伤风感冒 ”，我们想找到原因，以便对症下药。可是，Windows 的窗户并没有打开，我们无法透过窗户看看 Windows 的内部世界到底是什么样的，这让我们困惑，尤其让喜欢追根寻源的人们感到失望。

Linux 带着一股清新的风翩翩而来，它并不成熟，也不完美，甚至自身有很多缺点，可 Internet 的龙卷风把它吹遍世界，世界各地的计算机爱好者狂热地喜欢上 Linux。Linux 不再是一个孤单的个体，而成为软件发展史上的“ 自由女神 ”，很多的计算机高手和计算机爱好者为之倾其了极大的热情。它在迅速地成长，短短几年功夫，从一个摇摇晃晃的婴儿成长为脚步稳健的少年。这一切都源于什么？那就是 Linux 的创始人 Linus Torvalds 把 Linux 适时地放入到了 GNU 公共许可证下。

1.1　GNU 与 Linux 的成长

GNU 是自由软件之父 Richard Stallman 在 1984 年组织开发的一个完全基于自由软件的软件体系，与此相应的有一份通用公共许可证（General Public License，简称 GPL）。Linux 以及与它有关的大量软件是在 GPL 的推动下开发和发布的。

自由软件之父 Stallman 像一个神态庄严的传教士一样喋喋不休地到处传播自由软件的福音，阐述他创立 GNU 的梦想：“ 自由的思想，而不是免费的午餐 ”。这位自由软件的“ 顶级神甫 ”为自己的梦想付出了大半生的努力，他不但自己创作了许多自由软件如 GCC 和 GDB，在他的倡导下，目前人们熟悉的一些软件如 BIND、Perl、Apache、TCP/IP 等都成了自由软件的经典之作。

如果说 Stallman 创立并推动了自由软件的发展，那么，Linus 毫不犹豫奉献给 GNU 的 Linux，则把自由软件的发展带入到一个全新的境界。

实际上，Linus 是一个理想主义者，但他又脚踏实地。当 Linux 的第一个“ 产品 ”版本

Linux 1.0 问世的时候，是按完全自由扩散版权进行扩散的。他要求 Linux 内核的所有源代码必须公开，而且任何人均不得从 Linux 交易中获利。他这种纯粹的自由软件的理想实际上妨碍了 Linux 的扩散和发展,因为这限制了 Linux 以磁盘拷贝或者 CD-ROM 等媒体形式发行的可能，也限制了一些商业公司参与 Linux 的进一步开发并提供技术支持的良好愿望。于是 Linus 决定转向 GPL 版权，这一版权除了规定自由软件的各项许可权之外，还允许用户出售自己的程序拷贝。

这一版权上的转变对 Linux 的进一步发展可谓至关重要。从此以后，便有很多家技术力量雄厚又善于市场运作的商业软件公司，加入到了原先完全由业余爱好者和网络黑客所参与的这场自由软件运动，开发出了多种 Linux 的发行版本，磨光了自由软件许多不平的棱角，增加了更易于用户使用的图形用户界面和众多的软件开发工具，这极大地拓展了 Linux 的全球用户基础。

Linux 内核的功能以及它和 GPL 的结合，使许多软件开发人员相信这是有前途的项目，开始参加内核的开发工作。并将 GNU 项目的 C 库、gcc、Emacs、bash 等很快移植到 Linux 内核上来。可以说，Linux 项目一开始就和 GNU 项目紧密结合在一起，系统的许多重要组成部分直接来自 GNU 项目。Linux 操作系统的另一些重要组成部分则来自加利福尼亚大学 Berkeley 分校的 BSD UNIX 和麻省理工学院的 X Window 系统项目。这些都是经过长期考验的成果。

正是 Linux 内核与 GNU 项目、BSD UNIX 以及 MIT 的 X11 的结合，才使整个 Linux 操作系统得以很快形成，而且建立在稳固的基础上。

当 Linux 走向成熟时，一些人开始建立软件包来简化新用户安装和使用 Linux。这些软件包称为 Linux 发布或 Linux 发行版本。发行 Linux 不是某个个人或组织的事。任何人都可以将 Linux 内核和操作系统其他组成部分组合在一起进行发布。在早期众多的 Linux 发行版本中，最有影响的是 Slackware 发布。当时它是最容易安装的 Linux 发行版本，在推广 Linux 的应用中，起了很大的作用。Linux 文档项目（LDP）是围绕 Slackware 发布写成的。目前，RedHat 发行版本的安装更容易，应用软件更多，已成为最流行的 Linux 发行版本；而 Caldera 则致力于 Linux 的商业应用，它的发展速度也很快。这两个发行版本也有相应的成套资料。在中文的 Linux 发行版本方面，国内已经有众多的 Linux 厂商，如红旗 Linux，BluePoint Linux，中软 Linux 等。每种发行版本有各自的优点和弱点，但它们使用的内核和开发工具则是一致的。

1.2　Linux 的开发模式和运作机制

自由软件的出现，改变了传统的以公司为主体的封闭的软件开发模式。采用了开放和协作的开发模式，无偿提供源代码，允许任何人取得、修改和重新发布自由软件的源代码。这种开发模式激发了世界各地的软件开发人员的积极性和创造热情。大量软件开发人员投入到自由软件的开发中。软件开发人员的集体智慧得到充分发挥，大大减少了不必要的重复劳动，并使自由软件的脆弱点能够及时发现和克服。任何一家公司都不可能投入如此强大的人力去开发和检验商品化软件。这种开发模式使自由软件具有强大的生命力。

商业 UNIX 开发过程中，整个系统的开发有严格的质量保证措施、完整的文挡、完善的

源代码、全面的测试报告及相应的解决方案。开发者不能随意增加程序的特性和修改代码的关键部分，如果要修改代码，他们得将其写入错误报告中才能使其有效，并随后接收源代码控制系统的检查，如果发现修改不合适，修改也可能作废。每个开发者设计系统代码的一个或几个部分，开发者只有在程序检查过程中才能更改相应的代码。质量保证部门在内部对新的操作系统进行严格的回归测试，并报告发现的问题，开发者则有责任解决所报告的问题。质量保证部门采用复杂的统计分析系统以确保在下次发行时有百分之几的程序错误已修改。

总之，商业 UNIX 开发过程使得其代码非常复杂，公司为了保证下次操作系统的修订质量，得收集和统计分析操作系统的性能。开发商业 UNIX 是一个很大的工程，常常大到有数以百计的编程者、测试员、文档员以及系统管理员参与。

对于 Linux，你可将整个组织开发的概念、源代码控制系统、结构化的错误报告、统计分析等通通扔到一边去。

Linux 最初是由一群来自世界各地的自愿者通过 Internet 共同进行开发的。通过互联网和其他途径，任何人都有机会辅助开发和调试 Linux 的内核、链接新的软件、编写文档或帮助新用户。实际上，并没有单独的组织负责开发此系统，Linux 团体大部分通过邮递清单和 USENET 的消息组进行通信。许多协定已跳过开发过程，如果你想将自己的代码包括进“正式”内核，只需给 Linus Torvalds 发一个邮件，他就会进行测试并将其包括进内核（只要代码不使内核崩溃并且不与整个系统设计相悖，Linus 都很乐意将其包括进去）。

Linux 系统本身采用彻底开放、注重特性的方法进行设计。一般规律是大约隔几个月就发行一个 Linux 内核的新版本。当然发行周期还依赖于其他一些因素，如排除的程序故障数、用户测试预发行版的返回数以及 Linux 的工作量等。

可以说在两次发行期间，并不是每个故障都已排除，每个问题都已得到了解决。只要系统不出现很挑剔或明显的故障，就认为比较稳定，可以推出新版本。Linux 开发的动力不在于追求完美、无故障，而是要开发 UNIX 的免费实现。

如果你想把新的特性或应用软件增加到系统上，就得经过一个“初始”阶段。所谓“初始”阶段，就是一个由一些想对新代码挑出问题的用户不断进行测试的阶段。由于 Linux 团体大多在 Internet 上，“初始”软件通常安装在一个或多个 LinuxFTP 上，并且在 LinuxUSENET 消息组上张贴一张如何获取和测试其代码的消息，从而使得下载和测试“初始”软件的用户可以将结果、故障或问题等邮件告之作者。

初始代码中的问题解决后，代码就进入“第二”阶段：工作稳定但还不完全（即能够工作，但可能还不具备所有特性）。当然，它也可能进入“最后”阶段，即软件已完备并且可以使用。对于内核代码，一旦它完备，开发者就可让 Linus 将其包括进标准内核内，或者作为内核的可增加选项。

注意，这些仅是达成协定，并未形成规则。很多人对他们的软件不必发行“初始”或测试版充满信心，因此发行哪个版本是根据开发者的决定而定的。

你可能对一群自愿者居然能编写、调试出完整的 UNIX 系统惊讶不已。整个 Linux 内核通过拼凑而成，没有采用专利的源代码，大量工作都由自愿者完成，他们将 GNU 下的免费软件移植到 Linux 系统下，同时开发出库、文件系统以及通用的设备硬件驱动程序等。

实际上，Linus 率领的分布在世界各地的 Linux 内核开发队伍仍然在高速向前推进。当前推出的稳定的 Linux 内核的 2.4.x 版本充分显示了 Linux 开发队伍的非凡的创造能力以及

协作开发模式的价值。

1.3 走进 Linux 内核

如果说 CPU 是计算机硬件的心脏的话，那么，操作系统的内核则是整个计算机系统的心脏，或者说，是最高管理机构。Linux 的内核包含些什么？简单地说，它包含五大部分内容：进程调度、内存管理、进程间通信、虚拟文件系统及网络接口这五部分，我们也称为五个子系统。在走进 Linux 内核前，读者可能想知道，它到底有什么特点呢？

1.3.1 Linux 内核的特征

Linux 是个人计算机和工作站上的类 UNIX 操作系统。但是，它绝不是简化的 UNIX。相反，Linux 是强有力和具有创新意义的类 UNIX 操作系统。它不仅继承了 UNIX 的特征，而且在许多方面超过了 UNIX。作为类 UNIX 操作系统，Linux 内核具有下列基本特征。

（1）Linux 内核的组织形式为整体式结构。也就是说整个 Linux 内核由很多过程组成，每个过程可以独立编译，然后用连接程序将其连接在一起成为一个单独的目标程序。从信息隐藏的观点看，它没有任何程度的隐藏——每个过程都对其他过程可见。这种结构的最大特点是内部结构简单，子系统间易于访问，因此内核的工作效率较高。另外，基于过程的结构也有助于不同的人参与不同过程的开发，从这个角度来说，Linux 内核又是开放式的结构，它允许任何人对其进行修正、改进和完善。

（2）Linux 的进程调度方式简单而有效。可以说 Linux 在追求效率方面孜孜不倦，体现在调度方式上也是别具一格。对于用户进程，Linux 采用简单的动态优先级调度方式；对于内核中的例程（如设备驱动程序、中断服务程序等）则采用了一种独特的机制——软中断机制，这种机制保证了内核例程的高效运行。

（3）Linux 支持内核线程（或称守护进程）。内核线程是在后台运行而又无终端或登录 shell 和它结合在一起的进程。有许多标准的内核线程，其中有一些周期地运行来完成特定的任务（如 swapd），而其余一些则连续地运行，等待处理某些特定的事件（如 inetd 和 lpd）。内核线程可以说是用户进程，但和一般的用户进程又有不同，它像内核一样不被换出，因此运行效率较高。

（4）Linux 支持多种平台的虚拟内存管理。内存管理是和硬件平台密切相关的部分，为了支持不同的硬件平台而又保证虚拟存储管理技术的通用性，Linux 的虚拟内存管理为不同的硬件平台提供了统一的接口，因此把 Linux 内核移植到一个新的硬件平台并不是一件很困难的事。

（5）Linux 内核另一个独具特色的部分是虚拟文件系统（VFS Virtul File Systen）。虚拟文件系统不仅为多种逻辑文件系统（如 ext2,fat 等）提供了统一的接口，而且为各种硬件设备（作为一种特殊文件）也提供了统一接口。

（6）Linux 的模块机制使得内核保持独立而又易于扩充。模块机制可以使内核很容易地增加一个新的模块（如一个新的设备驱动程序），而无需重新编译内核；同时，模块机制还

可以把一个模块按需添加到内核或从内核中卸下，这使得我们可以按需要定制自己的内核。

（7）增加系统调用以满足特殊的需求。一般来说，系统调用是操作系统的设计者提供给用户使用内核功能的接口，但 Linux 开放的源代码也允许你设计自己的系统调用，然后把它加入到内核。

（8）网络部分面向对象的设计思想使得 Linux 内核支持多种协议、多种网卡驱动程序变得容易。

1.3.2 Linux 内核版本的变化

自从 1991 年 9 月 17 日，Linus Torvals 正式宣布 Linux 的第一个正式版本——0.02 版本，到现在，Linux 的内核版本发生了一系列的变化，新旧版本之间的时间间隔为几个月甚至几个星期，关于这一变化的非常详细的资料请看站点 http://ps.cus.umist.ac.uk/~rhw/kernel. versions.html 的内容。

我们把内核版本之间内容较大的变化分为三个阶段，第一阶段为 0.02～0.99.15j，第二阶段为 1.0～1.2.x，第三阶段为 1.2.x～2.x.x。一般来说，一个软件要到理论上已经完备或者已经没有毛病时才给予 1.0 版本的版本号，而 Linux 2.0 以后的版本比起 1.2.x 版本有了较大幅度的变化，请看站点 http://www.linuxhq.com/的内容。

从 Linux 诞生开始，Linux 内核就从来没有停止过升级，从 Linus 第一次发布的 0.02 版本到 1999 年具有里程碑意义的 2.2 版本，一直到现在的 2.4 版本，都凝聚了 Linux 内核开发人员大量辛苦的劳动。目前 Linux 在各种工作平台上，包括企业服务器和个人电脑上的广泛应用，使得 Linux 成为了 Windows 的强劲对手。

本书所分析的 Linux 内核版本是 2.4 版的 2.4.16 版。那么 Linux 2.4 版具有什么样的特点呢，我们可以用四个字来概括，那就是“广、新、快、小”。

1．广泛的支持

- 处理器芯片的广泛支持：Linux 2.4 提供了大量的处理器芯片的支持。原先的 Linux 就可以支持多种处理器体系结构，如 Intel x86、Motorola/IBM PowerPC、Compaq（DEC）Alpha 等，现在还增加了对 IA 64、S/390、SuperH 这 3 种体系结构的处理器的支持。对 Intel 的 x86 系列来说，AMD 和 Cyrix 公司的系列处理器产品也是使用 x86 指令的，同样也能获得很好的支持。
- 对 ISA 即插即用设备的支持 过去在 Linux 核心开发小组里面存在有两种不同的观点，一种是支持对 ISA 即插即用，另外一种持反对意见，认为对即插即用的支持简直是多余的。因此过去在 Linux 里对即插即用设置的通用做法只能是利用用户级的工具（如 isapnp tools），手动配置即插即用设备。现在的内核则有所不同了，在内核级实现了对即插即用的管理。我们可以看到系统会在启动的时候自动完成对即插即用设备的检测和自动配置，比如说，我们可以从一个即插即用的 IDE 控制器上启动系统。
- 广泛的文件系统支持：很少有一个操作系统能支持这么多种文件系统。Linux 使用的是 VFS（虚拟文件系统）的技术，提供了对多种文件系统的支持。从 Linux 1.x 到 Linux 2.2，Linux 已经可以支持多种文件系统。如 Windows 9x 的 VFAT、DOS 的 FAT、Mac OS 的 HFS、OS/2

的 HPFS、Windows NT 的 NTFS（NTFS 的支持还处于测试阶段）等；当然还包括 Linux 自己使用的高性能的 Ext2 文件系统。新版本的 Linux 新增支持现在的 DVD 使用的 UDF 文件系统和 SGI 的 IRIX 系统上的 XFS 文件系统。

在 Windows 里面使用 SMB 协议来实现“网上邻居”的共享访问，Linux 2.4 的内核里会让您自己选择是否从 Windows 98/NT 下载驱动器，还可以自动检测远端的系统类型，使得 Linux 在 Windows 环境的局域网里工作得更好。

对 NFS（网络文件系统）来说，Linux 2.4 版本支持最近发布的 NFS v3 版本的网络文件系统。

- 对软猫的支持：软猫实际上也被称为 WinModem，就是因为现有的这种软猫的驱动都是由为 Windows 开发的软件来完成的。这种 Modem 和一般 Modem 的处理方法不同，它的 DSP 处理并不是在硬件层次上完成的，而是使用软件通过 CPU 来实现的，因此无法在现有的 Linux 中配置这种 Modem 上网。现在的 Linux 内核里已经开始了这方面的支持。

2．新思路

- 新型的设备管理方法：Linux 2.4 引入了 I2O（Intelligent Input/Output）的设备驱动管理方法。它的做法是，将驱动程序分成了两个部分：一个是在操作系统模块的部分，另外一个是在硬件模块的部分。操作系统模块的部分是独立的，硬件模块的部分是依赖于硬件结构的。这种新型的管理方法使得 Linux 2.4 可以更好地支持大部分的 ISA 和 PCI 设备。
- 对 USB 总线的支持：近年来，USB（通用串口总线）的技术是计算机界振奋人心的事情之一，现在已经出现了大量的使用这种接口的设备，如键盘、鼠标、音箱、Modem 等。使用 USB 接口使得计算机外设的安装和使用变得更为简单，自然成为了一种潮流。现在的 Linux 也可以很好地支持这种总线接口的设备。
- 新型的二进制执行代码类型（Binary Types）：Linux 是第一个在内核级提供内建 Java 解释器的支持，从而进行 Java 代码的执行的操作系统之一。这在 Linux 2.2 版本里已经实现了。Linux 2.4 版本又做了改进，将这种支持的方法改为对“Misc”二进制类型的支持。通过使用这种类型的二进制代码类型，用户甚至可以利用 DOSEMU（MS DOS 模拟器）或者 WINE（MS Windows 模拟器）来运行在 DOS/Windows 下的.exe 或.com 的程序。同样用户也可以自己配置出 Java 字节码运行类型。
- 内核级的 Web 服务器：这种 Web 服务器和所谓的 Apache 用户层上的 Web 服务器并不冲突。对 HTTP 请求首先由内核级的 Web 服务器进行处理，如果不能处理就将请求提交给 Apache 用户级 Web 服务器来处理。像这样的构思和实现在网络操作系统中实属一绝。

3．高性能

- 对虚拟文件系统（VFS）的修改：Linux 2.4 版本的文件系统修改了 VFS 中的错误，尤其是在文件的缓存管理上。过去的文件系统的高速缓存管理是建立在复杂的双缓冲池（dual-buffer pool）上的，这种方法导致连开发人员都不知道什么时候将双缓冲池进行同步。这种处理方法并没有给文件处理带来好处，反而增加了内存的使用。因为要处理双缓冲系统的同步，使得系统的处理速度降低。现在开发人员修改这段代码，使用了简单有效的单缓冲系统，提高了文件系统的处理效率

- 对高端服务器的支持：Linux 2.4 版本的内核可以支持在 SMP（对称多处理器系统）下的多个 IO-APIC（输入输出的高级可编程中断控制器），提高了对高端服务器的支持效率。

Linux 2.4 版本可以支持多达 10 个 IDE 控制器。过去的 Linux 版本只能支持最多 4 个 ID 控制器。一些强大的企业级 Web 服务器正需要这样的硬件支持。

Linux 2.4 版本可以支持 Intel P6 以上芯片的 MTRR（内存类型范围寄存器），对非 Intel 的如 Cyrix 6x86、6x86MX、MII 的 ARR（地址范围寄存器）也能有很好的支持，这使一些高带宽的设备的运行性能得到了提高。

现在的内核可以支持多达 42 亿个用户。在 Intel 架构上可以支持到多达 4GB 的内存。并且现在的内核还可以支持多达 16 块以太网卡，同时支持最大容量为 2GB 的文件。

这些性能都使得 Linux 对高端设备的支持能力得到了提高。

- 对高速网的支持：Linux 2.4 版本支持 ATM 网络适配器等高速网络设备，为进一步的网络发展做好了准备。对低端用户来说，Linux 提供的 PPP 层和 ISDN 层的结合，提供了在并口线上的 PPP 和在以太网上的 PPP 支持。

4．小内核

- 内核本来就很小：Linux 的整个内核源代码大概需要占用 20 多 MB 的硬盘空间，但是编译出来的二进制代码只占用 600KB 左右的空间，完全可以放在一张软盘上，随时可以使用这张软盘启动系统。
- 对内存的需求很小：大家比较关心的一个问题是 Linux 现在需要多少内存才能正常工作。我们知道，大部分的操作系统在升级的同时，对硬件的需求也在不断提高，尤其是对内存的需求方面，很大层次上影响了系统的性能。不过 Linux 和其他操作系统不同，Linux 可以进行个性化的定制，用户完全可以根据自己的系统配置来生成自己需要的操作系统内核，也可以根据需要启动或关闭一些系统服务，这样可以减少系统对资源的占用，提高系统的运行效率。

Linux 内核发展到现在已经相当庞大，要想在一段时间内搞清所有的内容几乎是不可能的，因此，本书对内核的分析也集中在几个主要部分的主要内容上，其运行的平台也只选择了 i386 的单 CPU，在一些特殊情况下，我们也会讨论 SMP（对称多处理机）的情况。

走进 Linux 不是一件容易的事，但走出来同样不容易。阅读 Linux 源代码如同阅读一篇优美的作品，会深深地吸引着你，既可以满足你好奇的愿望，也可以检验你挑战困难的勇气。

1.4　分析 Linux 内核的意义

Linux 开放的源代码为我国软件产业的发展和腾飞提供了前所未有的机遇，这体现在以下几个方面。

1.4.1 开发适合自己的操作系统

因为操作系统是所有软件赖以生存的基础，因此，我们强烈地需要拥有自己的操作系统，这不仅对我们国家的民族软件发展有极大的好处，而且对国家的安全和国防事业都至关重要。但是如果象日本那样搞自己的一套体系结构（PC98），不与国际标准兼容，结果会严重阻碍软件业的发展，那也是死路一条。但是国产操作系统没有任何市场，而 Windows 又几乎处于垄断地位，面对这种局面，出路何在？Linux 的出现正符合我们所有的要求，因为源代码公开，我们可以立即加入开发，不仅开发速度大大快于任何商业操作系统，并且可以保证操作系统中不存在任何黑洞和隐蔽的问题，永远不会受制于人。因为 Linux 是国际化的，我们也不必考虑兼容性问题，永远不会同国际脱轨。因此 Linux 对于我们来说，是实现民族软件腾飞的一个难得的机遇。

实际上，操作系统的发展必将出现基于某一标准的百花齐放的局面，定制适合自己的操作系统也将不仅仅是梦想。但是，开发一个操作系统不是一件容易的事，甚至分析一个现有的操作系统也并不简单，而 Linux 作为分析实例是比较合适的。因为 Linux 的开放、众多人的参与以及 Linux 社区的互助都为 Linux 的学习和普及提供了良好的外部环境。

1．开发嵌入式操作系统

Linux 为嵌入操作系统提供了一个极有吸引力的选择，它和 UNIX 相似，是以内核为基础的、完全内存保护、多任务多进程的操作系统。支持广泛的计算机硬件，包括 X86、Alpha、Sparc、MIPS、PPC、ARM、NEC、MOTOROLA 等现有的大部分芯片。程序源码全部公开，任何人可以修改并在 GNU 通用公共许可证（GNU General Public License）下发行，这样，开发人员可以对操作系统进行定制，再也不必担心像 Windows 操作系统中“后门”的威胁。同时由于有 GPL 的控制，大家开发的东西大都相互兼容，不会走向分裂之路。Linux 用户遇到问题时可以通过 Internet 向网上成千上万的 Linux 开发者请教，这使最困难的问题也有办法解决。

正是嵌入式操作系统的特殊要求为 Linux 在嵌入式系统中的发展提供了广阔的空间，使得 Linux 成为嵌入式操作系统中的新贵。在应用上，嵌入式 Linux 可应用于信息家电（机顶盒、数字电视）、多媒体手机、工业、商业控制（智能工控设备、POS/ATM 机）、电子商务平台，甚至军事应用等。

2．开发实时操作系统

在实时 Linux 出现之前，在为实时应用选择系统平台的时候，人们大抵只有两种选择，要么使用 DOS 并自己编写所有必要的驱动程序，要么就得购买专用的实时系统。前者不仅费时费力，其性能也难以令人满意。而后者性能虽佳，其价格却高得让人难以接受。

实时 Linux 的出现解决了这一问题，它为实时应用领域的研究与开发提供了一个物美价廉的完备的操作系统平台。凭着自身的技术特色，借助于 Linux 的强大功能，实时 Linux 下开发出的实时应用有着不俗的表现。

1.4.2　开发高水平软件

自由软件联盟及“ 中国自由软件库 ”就已涵盖了操作系统、开发语言、视窗系统、数据库、网络、文字处理、排版及多媒体等各个领域，还有 VCD 解压源程序、路由器源程序等。利用自由软件让个人计算机带十几个硬盘实现阵列技术，及其亚微米超大规模集成电路 CAD 系统，可直接输出生产线控制数据等，这能让我们学到最先进的软件开发规范和开发技术，Linux 内核的许多面向通信的底层代码对开发我国自己的信息安全产品极有参考价值。

实际上，目前 Linux 的源代码中包含了世界各地几百名计算机高手的作品，分析这些源代码对于我们掌握核心技术会起到事半功倍的作用，尤其是各种驱动程序的编写，对于我们把软硬件结合起来发展民族信息产业至关重要。要改变目前我国软件开发在低层次上的重复过程，必须掌握操作系统的核心技术。

只要站在“ 巨人 ”的肩上，认真钻研，就一定能吃透它，利用它，研制出自己的解压芯片、路由器、磁盘阵列产品，开发出高级的 CAD 系统等，打破国外的技术封锁，振兴我国电子工业。

1.4.3　有助于计算机科学的教学和科研

对于从事计算机科学教学和科研的人来说，Linux 具有更多一层的意义。一般市场上出售的 UNIX，除了价格之外，还不提供其核心程序的源代码。这样，若想了解 UNIX 的内核，或在内核程序上作一些改进就很困难，更谈不上作为操作系统教学和科研的平台了，而 Linux 提供了从内核到上层的所有软件的全部源程序代码。在易于获得源代码的条件下，如果能对源代码的组织结构、实现原理及实现机制进行较详细的描述，那么对很多人深入了解源程序将有很大帮助。

实际上，Linux 也很适合教学用操作系统，一般的操作系统教材只讲操作系统的实现原理，学生既觉得抽象又感觉不到操作系统的重要价值。尽管有些书也是以 UNIX 为实例，但学生很难接触到 UNIX 操作系统，这对学生真正深入了解操作系统造成了困难。

国外很多大学已经把 Linux 作为教学用操作系统，我们认为这主要是因为：①Linux 平台易于建立；②Linux 内核源代码易于获得；③Linux 结构简单、清晰；④Linux 的实现采用了大量的数据结构，可以锻炼学生的抽象能力和知识应用能力。

可以说，Linux 内核源代码的开放乃至自由联盟各种应用程序源代码的开放，为我们的软件教学提供了活教材，我们的学生可以在这种“ 自由 ”文化的氛围下，学习并掌握软件开发的核心技术，我们就有希望在 21 世纪不仅拥有中文 Linux 操作系统，而且拥有适合中国国情的大量而优秀的 Linux 应用软件。

1.5　Linux 内核结构

Linux 是一个庞大、高效而复杂的操作系统，虽然它的开发起始于 Linus Torvalds 一

个人，但随着时间的推移，越来越多的人加入了 Linux 的开发和对它的不断完善。如何从整体上把握 Linux 内核的体系结构，对于 Linux 的开发者和分析者都至关重要。

1.5.1　Linux 内核在整个操作系统中的位置

Linux 的内核不是孤立的,必须把它放在整个系统中去研究,如图 1.1 所示,显示了 Linux 内核在整个操作系统的位置。

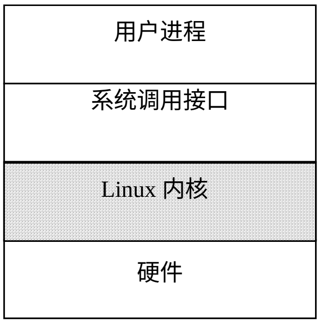


图 1.1　Linux 内核在整个操系统中的位置

由图 1.1 可以看出，Linux 操作系统由 4 个部分组成。

1．用户进程

用户应用程序是运行在 Linux 操作系统最高层的一个庞大的软件集合。当一个用户程序在操作系统之上运行时，它成为操作系统中的一个进程。

2．系统调用接口

在应用程序中，可通过系统调用来调用操作系统内核中特定的过程，以实现特定的服务。例如，在程序中安排一条创建进程的系统调用，则操作系统内核便会为之创建一个新进程。

系统调用本身也是由若干条指令构成的过程。但它与一般的过程不同，主要区别是：系统调用是运行在内核态（或叫系统态），而一般过程是运行在用户态。在 Linux 中，系统调用是内核代码的一部分。

3．Linux 内核

这是本书要讨论的重点。内核是操作系统的灵魂，它负责管理磁盘上的文件、内存，负责启动并运行程序，负责从网络上接收和发送数据包等。简言之，内核实际是抽象的资源操作到具体硬件操作细节之间的接口。

4．硬件

这个子系统包括了 Linux 安装时需要的所有可能的物理设备。例如，CPU、 内存、硬盘、网络硬件等。

上面的这种划分把整个 Linux 操作系统分为 4 个层次。把用户进程也纳入操作系统的范围内是因为用户进程的运行和操作系统密切相关，而系统调用接口可以说是操作系统内核的扩充，硬件则是操作系统内核赖以生存的物质条件。这 4 个层次的依赖关系表现为：上层依赖下层。

1.5.2　Linux 内核的作用

从程序员的角度来讲，操作系统的内核提供了一个与计算机硬件等价的扩展或虚拟的计算平台。它抽象了许多硬件细节，程序可以以某种统一的方式进行数据处理，而程序员则可以避开许多硬件细节。从另一个角度讲，普通用户则把操作系统看成是一个资源管理者，在它的帮助下，用户可以以某种易于理解的方式组织自己的数据，完成自己的工作并和其他人共享资源。

Linux 以统一的方式支持多任务，而这种方式对用户进程是透明的，每一个进程运行起来就好像只有它一个进程在计算机上运行一样，独占内存和其他的硬件资源，而实际上，内核在并发地运行几个进程，并且能够让几个进程公平合理地使用硬件资源，也能使各进程之间互不干扰安全地运行。

1.5.3　Linux 内核的抽象结构

Linux 内核由 5 个主要的子系统组成，如图 1.2 所示。

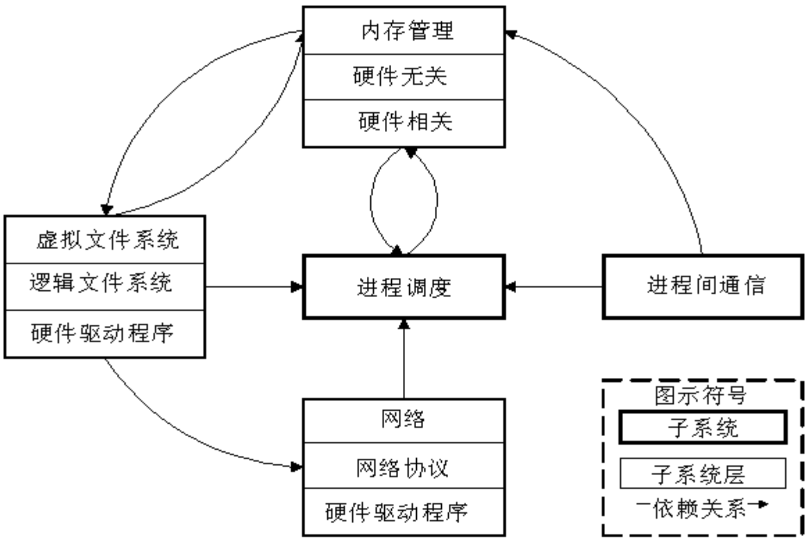


图 1.2　Linux 内核子系统及其之间的关系

(1) 进程调度（SCHED）控制着进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际是仅等待 CPU 资源的进程，如果某个进程在等待其他资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

(2) 内存管理（MM）允许多个进程安全地共享主内存区域。Linux 的内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据和堆栈的总量可以超过实际内存的大小，操

作系统只将当前使用的程序块保留在内存中，其余的程序块则保留在磁盘上。必要时，操作系统负责在磁盘和内存之间交换程序块。

内存管理从逻辑上可以分为硬件无关的部分和硬件相关的部分。硬件无关的部分提供了进程的映射和虚拟内存的对换；硬件相关的部分为内存管理硬件提供了虚拟接口。

（3）虚拟文件系统（Virtul File System，VFS）隐藏了各种不同硬件的具体细节，为所有设备提供了统一的接口，VFS 还支持多达数十种不同的文件系统，这也是 Linux 较有特色的一部分。

虚拟文件系统可分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2，fat 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

（4）网络接口（NET）提供了对各种网络标准协议的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

（5）进程间通信（IPC） 支持进程间各种通信机制。从图 1.2 所示可以看出，处于中心位置的是进程调度，所有其他的子系统都依赖于它，因为每个子系统都需要挂起或恢复进程。一般情况下，当一个进程等待硬件操作完成时，它被挂起；当操作真正完成时，进程被恢复执行。例如，当一个进程通过网络发送一条消息时，网络接口需要挂起发送进程，直到硬件成功地完成消息的发送，当消息被发送出去以后，网络接口给进程返回一个代码，表示操作的成功或失败。其他子系统（内存管理，虚拟文件系统及进程间通信）以相似的理由依赖于进程调度。

各个子系统之间的依赖关系如下。

- 进程调度与内存管理之间的关系：这两个子系统互相依赖。在多道程序环境下，程序要运行必须为之创建进程，而创建进程的第一件事，就是要将程序和数据装入内存。
- 进程间通信与内存管理的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，这种机制允许两个进程除了拥有自己的私有内存，还可存取共同的内存区域。
- 虚拟文件系统与网络接口之间的关系：虚拟文件系统利用网络接口支持网络文件系统（NFS），也利用内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程（swapd）定期地由调度程序调度，这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前正在运行的进程。

除了如图 1.2 所示的依赖关系以外，内核中的所有子系统还要依赖一些共同的资源，但在图中并没有显示出来。这些资源包括所有子系统都用到的过程，例如分配和释放内存空间的过程，打印警告或错误信息的过程，还有系统的调试例程等。

1.6 Linux 内核源代码

为了深入地了解 Linux 的实现机制，还必须阅读 Linux 的内核源代码，下面是对有关源

代码的介绍。

1.6.1　多版本的内核源代码

对不同的内核版本，系统调用一般是相同的。新版本也许可以增加一个新的系统调用，但旧的系统调用将依然不变，这对于保持向后兼容是非常必要的——一个新的内核版本不能打破常规的过程。在大多数情况下，设备文件将仍然相同，而另一方面，版本之间的内部接口有所变化。

Linux 内核源代码有一个简单的数字系统，任何偶数内核（如 2.0.30）是一个稳定的版本，而奇数内核（如 2.1.42）是正在发展中的内核。本书是基于稳定的 2.4.16 源代码的。发展中的内核总是有最新的特点，支持最新的设备，尽管它们还不稳定，也许不是你所想要的，但它们是发展最新而又稳定的内核的基础。

目前，较新而又稳定的内核版本是 2.2.x 和 2.4.x，因为版本之间稍有差别，因此，如果你想让一个新驱动程序模块既支持 2.2.x，也支持 2.4.x，就需要根据内核版本对模块进行条件编译。

对内核源代码的修改是以补丁文件的形式发布的。patch 实用程序用来对内核源文件进行一系列的修订，例如，如果你有 2.4.9 内核源代码，而想移到 2.4.16，你可以获得 2.4.16 的补丁文件，应用 patch 来修订 2.4.9 源文件。例如：

```
$ cd /usr/src/linux
$ patch -p1 < patch-2.4.16
```

1.6.2　Linux 内核源代码的结构

Linux 内核源代码位于/usr/src/linux 目录下，其结构分布如图 1.3 所示，每一个目录或子目录可以看作一个模块，其目录之间的连线表示“子目录或子模块”的关系。下面是对每一个目录的简单描述。

include/目录包含了建立内核代码时所需的大部分包含文件，这个模块利用其他模块重建内核。

init/ 子目录包含了内核的初始化代码，这是内核开始工作的起点。

arch/子目录包含了所有硬件结构特定的内核代码，如图 1.3 所示，arch/子目录下有 i386 和 alpha 模块等。

drivers/ 目录包含了内核中所有的设备驱动程序，如块设备，scsi 设备驱动程序等。

fs/ 目录包含了所有文件系统的代码，如：ext2，vfat 模块的代码等。

net/ 目录包含了内核的连网代码。

mm/ 目录包含了所有的内存管理代码。

ipc/ 目录包含了进程间通信的代码。

kernel/　目录包含了主内核代码。

图 1.3 显示了 8 个目录，即 init、kernel、mm、ipc、drivers、fs、arch 及 net 的包含文件都在“include/”目录下。在 Linux 内核中包含了 drivers、fs、arch 及 net 模块，

这就使得 Linux 内核既不是一个层次式结构，也不是一个微内核结构，而是一个“整体式”结构。因为系统调用可以直接调用内核层，因此，该结构使得整个系统具有较高的性能，其缺点是内核修改起来比较困难，除非遵循严格的规则和编码标准。

在图 1.3 中所示的模块结构，代表了一种工作分配单元。利用这种结构，我们期望 Linus Torvalds 能维护和增强内核的核心服务，即 init/、kernel/、mm/及 ipc/，其他的模块 drivers、fs、arch 及 net 也可以作为工作单元，例如，可以分配一组人对块文件系统进行维护和进一步地开发，而另一组人对 scsi 文件系统进行完善。图 1.3 所示类似于 Linux 的自愿者开发队伍一起工作来增强和扩展整个系统的框架。

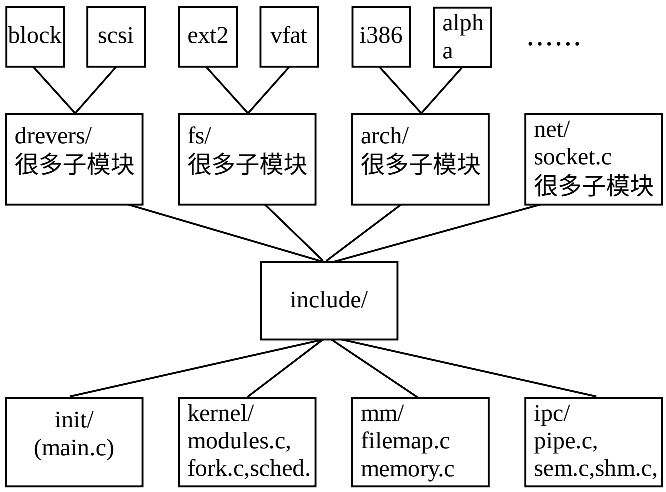


图 1.3 Linux 源代码的分布结构

1.6.3 从何处开始阅读源代码

像 Linux 内核这样庞大而复杂的程序看起来确实让人望而生畏，它像一个很大的球，没有起点和终点。在读源代码的过程中，你会遇到这样的情况，当读到内核的某一部分时又会涉及到其他更多的文件，当返回到原来的地方想继续往下读时，又忘了原来读的内容。在 Internet 上，很多人为此付出了很大的努力，制作出了源代码导航器，这为源代码阅读提供了很好的条件，下载站点为http://lxr.linux.no/source。下面给出阅读源代码的一些线索。

1．系统的启动和初始化

在基于 Intel 的系统上，当 loadlin.exe 或 LILO 把内核装入到内存并把控制权传递给内核时，内核开始启动。关于这一部分，看 arch/i386/kernel/head.S，head.S 进行特定结构的设置，然后跳转到 init/main.c 的 main（）例程。

2．内存管理

内存管理的代码主要在/mm，但特定结构的代码在 arch/*/mm。缺页中断处理的代码在

mm/memory.c ，而内存映射和页高速缓存器的代码在 mm/filemap.c。缓冲器高速缓存是在 mm/buffer.c 中实现，而交换高速缓存是在 mm/swap_state.c 和 mm/swapfile.c 中实现。

3．内核

内核中，特定结构的代码在 arch/*/kernel，调度程序在 kernel/sched.c，fork 的代码在 kernel/fork.c，task_struct 数据结构在 include/linux/sched.h 中。

4．PCI

PCI 伪驱动程序在 drivers/pci/pci.c ，其定义在 include/linux/pci.h。每一种结构都有一些特定的 PCI BIOS 代码，Intel 的在 arch/alpha/kernel/bios32.c。

5．进程间通信

所有 System V IPC 对象权限都包含在 ipc_perm 数据结构中，这可以在 include/linux/ipc.h 中找到 System V 消息是在 ipc/msg.c 中实现，共享内存在 ipc/shm.c 中，信号量在 ipc/sem.c 中，管道在 ipc/pipe.c 中实现。

6．中断处理

内核的中断处理代码是几乎所有的微处理器所特有的。中断处理代码在 arch/i386/kernel/irq.c 中，其定义在 include/asm-i386/irq.h 中。

7．设备驱动程序

Linux 内核源代码的很多行是设备驱动程序。Linux 设备驱动程序的所有源代码都保存在/driver，根据类型可进一步划分为：

/block

块设备驱动程序如 ide（在 ide.c）。如果想看包含文件系统的所有设备是如何被初始化的，应当看 drivers/block/genhd.c 中的 device_setup（），device_setup（）不仅初始化了硬盘，当一个网络安装 nfs 文件系统时，它也初始化网络。块设备包含了基于 IDE 和 SCSI 的设备。

/char

这是看字符设备（如 tty，串口及鼠标等）驱动程序的地方。

/cdrom

Linux 的所有 CDROM 代码都在这里，如在这儿可以找到 Soundblaster CDROM 的驱动程序。注意 ide CD 的驱动程序是 ide-cd.c，放在 drivers/block；SCSI CD 的驱动程序是 scsi.c，放在 drivers/scsi。

/pci

这是 PCI 伪驱动程序的源代码，在这里可以看到 PCI 子系统是如何被映射和初始化的。

/scsi

在这里可以找到所有的 SCSI 代码及 Linux 所支持的 scsi 设备的所有设备驱动程序。

/net

在这里可以找到网络设备驱动程序，如 DECChip 21040 PCI 以太网驱动程序在 tulip.c

中。

/sound

这是所有声卡驱动程序的所在地。

8．文件系统

EXT2 文件系统的源代码全部在 fs/ext2/ 目录下，而其数据结构的定义在 include/linux/ ext2_fs.h,ext2_fs_i.h 及 ext2_fs_sb.h 中。虚拟文件系统的数据结构在 include/linux/fs.h 中描述，而代码是在 fs/*中。缓冲区高速缓存与更新内核的守护进程的实现是在 fs/buffer.c 中。

9．网络

网络代码保存在/net 中，大部分的 include 文件在 include/net 下，BSD 套节口代码在 net/socket.c 中，IP 第 4 版本的套节口代码在 net/ipv4/af_inet.c。一般的协议支持代码（包括 sk_buff 处理例程）在 net/core 下，TCP/IP 联网代码在 net/ipv4 下，网络设备驱动程序在/drivers/net 下。

10．模块

内核模块的代码部分在内核中，部分在模块包中，前者全部在 kernel/modules.c 中，而数据结构和内核守护进程 kerneld 的信息分别在 include/linux/module.h 和 include/linux/kerneld.h 中。如果想看 ELF 目标文件的结构，它位于 include/linux/elf.h 中。

1.7 Linux 内核源代码分析工具

凡是尝试做过内核分析的人都知道，Linux 的内核组织结构虽然非常有条理，但是，它毕竟是众人合作的结果，在阅读代码的时候要将各个部分结合起来，确实是件非常困难的事情。因为在内核中的代码层次结构肯定分多个层次，那么对一个函数的分析，肯定会涉及到多个函数，而对每一个函数都可能有多层的调用，一层层下来，直接在代码文件中查找肯定会让你失去耐心和兴趣的。最后，很多人只能望洋兴叹，或者只是找一本书来看看基本原理，却不敢去说自己真正看过内核源代码。任何一本原理书只能是你阅读源代码的导航器，绝不能代替源代码的阅读。

俗话说：“工欲善其事，必先利其器”。面对 Linux 这样庞大的源代码，必须有相应工具的支持才能使分析有效地进行下去。在此介绍两种源代码的分析工具，希望能对感兴趣的读者有所帮助。

1.7.1　Linux 超文本交叉代码检索工具

Linux 超文本交叉代码检索工具 LXR（Linux Cross Reference），是由挪威奥斯陆大学数学系 Arne Georg Gleditsch 和 Per Kristian Gjermshus 编写的。这个工具实际上运行在 Linux 或者 UNIX 平台下，通过对源代码中的所有符号建立索引，从而可以方便地检索任何一个符号，包括函数、外部变量、文件名、宏定义等。不仅仅是针对 Linux 源代码，对于 C 语言的其他大型的项目，都可以建立其 LXR 站点，以便开发者查询代码，以及后继开发者学习代码。

目前的 LXR 是专门为 Linux 下面的 Apache 服务器设计的，通过运行 perl 脚本，检索在安装时根据需要建立的源代码索引文件，将数据发送到网络客户端的 Web 浏览器上。任何一种平台上的 Web 浏览器都可以访问，这就方便了习惯在 Windows 平台下工作的用户。

LXR 的英文网站为 http://lxr.linux.no/，在中国 Linux 论坛 http://www.linuxforum.net上有其镜像。

读者如果想建立自己的LXR网站,则可直接通过http://lxr.linux.no/lxr-0.3.tar.gz，下载 LXR 的 tarball 形式的安装包。另外，因为 LXR 使用 glimpse 作为整个项目中文本的搜索工具，因此还需要下载 glimpse，位置在 http://glimpse.cs.arizona.edu ，下载 glimpse-4.12.6.bin.Linux- 2.2.5-22-i686.tar.gz，也可以使用更新的版本下载以后按照说明进行安装和配置，就可以建立自己的 LXR 网站。如果你上网很方便，就可以直接从 http://lxr.linux.no/网站查询所需要的各种源代码信息。

1.7.2　Windows 平台下的源代码阅读工具（Source Insight）

为了方便地学习 Linux 源程序，我们不妨回到我们熟悉的 Windows 环境下。但是在 Windows 平台上，使用一些常见的集成开发环境，效果也不是很理想，比如难以将所有的文件加进去，查找速度缓慢，对于非 Windows 平台的函数不能彩色显示。在 Windows 平台下有一个强大的源代码编辑器，它的卓越性能使得学习 Linux 内核源代码的难度大大降低，这便是 Source Insight 3.0，它是一个 Windows 平台下的共享软件，可以从 http://www.sourceinsight.com/上下载30天试用版本。由于Source Insight是一个Windows平台的应用软件，所以首先要通过相应手段把 Linux 系统上的程序源代码移到 Windows 平台下,这一点可以通过在 linux 平台上将/usr/src 目录下的文件拷贝到 Windows 平台的分区上，或者从网上或光盘中直接拷贝文件到 Windows 平台的分区上。

这个软件的安装非常简单，双击安装文件名，然后按提示进行即可。安装完成后，就可启动该程序。这个软件使用起来也非常简单：先选择 Project 菜单下的 new，新建一个工程，输入工程名，接着要求你把欲读的源代码加入（可以加入整个目录）后，该软件就分析你所加的源代码。分析完后，就可以进行阅读了。对于打开的阅读文件，如果想看某一变量的定义，先把光标定位于该变量，然后单击工具条上的相应选项，该变量的定义就显示出来。对于函数的定义与实现也可以同样操作。

第二章　Linux 运行的硬件基础

我们知道，操作系统是一组软件的集合。但它和一般软件不同，因为它是充分挖掘硬件潜能的软件，也可以说，操作系统是横跨软件和硬件的桥梁。因此，要想深入解析操作系统内在的运作机制，就必须搞清楚相关的硬件机制。

操作系统的设计者必须在硬件相关的代码与硬件无关的代码之间划出清楚的界限，以便将一个操作系统很容易地移植到不同的平台。Linux 的设计就做到了这点，它把与硬件相关的代码全部放在 arch（architecture 一词的缩写，即体系结构相关）目录下，在这个目录下，你可以找到 Linux 目前版本支持的所有平台，例如，Linux 2.4 支持的平台有 arm、alpha,、i386、m68k、mips 等十多种。在这众多的平台中，大家熟悉的就是 i386，即 Inte l80386。因此，我们所介绍的硬件基础也是以此为背景的。

在 X86 系列中，8086 和 8088 是 16 位的处理器，而从 80386 开始为 32 位处理器。这种变化看起来是处理器位数的变化，但实质上是处理器体系结构的变化，从寻址方式上说，就是从“ 实模式 ”到“ 保护模式 ”的变化。从 80386 以后，Intel 的 CPU 经历了 80486、Pentium、Pentium II、Pentium III 等型号，虽然它们在速度上提高了好几个数量级，功能上也有不少改进，但基本上属于同一种系统结构的改进与加强，而无本质的变化。因此，我们用 i386 统指这些型号。

2.1　i386 的寄存器

80386 作为 80X86 系列中的一员，必须保证向后兼容，也就是说，既要支持 16 位的处理器，又要支持 32 位的处理器。在 8086 中，所有的寄存器都是 16 位的，下面我们来看一下 780386 中寄存器有何变化。

- 把 16 位的通用寄存器、标志寄存器以及指令指针寄存器扩充为 32 位的寄存器
- 段寄存器仍然为 16 位。
- 增加 4 个 32 位的控制寄存器。
- 增加 4 个系统地址寄存器。
- 增加 8 个调式寄存器。
- 增加 2 个测试寄存器。

2.1.1　通用寄存器

8 个通用寄存器是 8086 寄存器的超集，它们的名称和用途分别为：

- EAX：一般用作累加器。
- EBX：一般用作基址寄存器（Base）。
- ECX：一般用来计数（Count）。
- EDX：一般用来存放数据（Data）。
- EBP：一般用作堆栈指针（Stack Pointer）。
- EBP：一般用作基址指针（Base Pointer）。
- ESI：一般用作源变址（Source Index）。
- EDI：一般用作目标变址（Destinatin Index）。

8 个通用寄存器中通常保存 32 位数据，但为了进行 16 位的操作并与 16 位机保持兼容，它们的低位部分被当成 8 个 16 位的寄存器，即 AX、BX……DI。为了支持 8 位的操作，还进一步把 EAX、EBX、ECX、EDX 这 4 个寄存器低位部分的 16 位，再分为 8 位一组的高位字节和低位字节两部分，作为 8 个 8 位寄存器。这 8 个寄存器分别被命名为 AH、BH、CH、DH 和 AL、BL、CL、DL。对 8 位或 16 位寄存器的操作只影响相应的寄存器。例如，在做 8 位加法运算时，位 7 的进位并不传给目的寄存器的位 9，而是把标志寄存器中的进位标志（CF）置位。因此，这 8 个通用寄存器既可以支持 1 位、8 位、16 位和 32 位数据运算，也支持 16 位和 32 位存储器寻址。

2.1.2 段寄存器

8086 中有 4 个 16 位的段寄存器：CS、DS、SS、ES，分别用于存放可执行代码的代码段、数据段、堆栈段和其他段的基地址。在 80386 中，有 6 个 16 位的段寄存器，但是，这些段寄存器中存放的不再是某个段的基地址，而是某个段的选择符（Selector）。因为 16 位的寄存器无法存放 32 位的段基地址，段基地址只好存放在一个叫做描述符表（Descriptor）的表中。因此，在 80386 中，我们把段寄存器叫做选择符。下面给出 6 个段寄存器的名称和用途。

- CS：代码段寄存器。
- DS：数据段寄存器。
- SS：堆栈段寄存器。
- ES、FS 及 GS：附加数据段寄存器。

有关段选择符、描述符表及系统表地址寄存器将在段机制一节进行详细描述。

2.1.3 状态和控制寄存器

状态和控制寄存器是由标志寄存器（EFLAGS）、指令指针（EIP）和 4 个控制寄存器组成，如图 2.1 所示。

1．指令指针寄存器和标志寄存器

指令指针寄存器（EIP）中存放下一条将要执行指令的偏移量（offset ），这个偏移量是相对于目前正在运行的代码段寄存器（CS）而言的。偏移量加上当前代码段的基地址，就形成了下一条指令的地址。EIP 中的低 16 位可以分开来进行访问，给它起名叫指令指针 IP

寄存器，用于 16 位寻址。

标志寄存器	EFLAGS
指令指针	EIP
机器状态字	CR0
Intel 预留	CR1
页故障地址	CR2
页目录地址	CR3

图 2.1　状态和控制寄存器

标志寄存器（EFLAGS）存放有关处理器的控制标志，如图 2.2 所示。标志寄存器中的第 1、3、5、15 位及 18～31 位都没有定义。

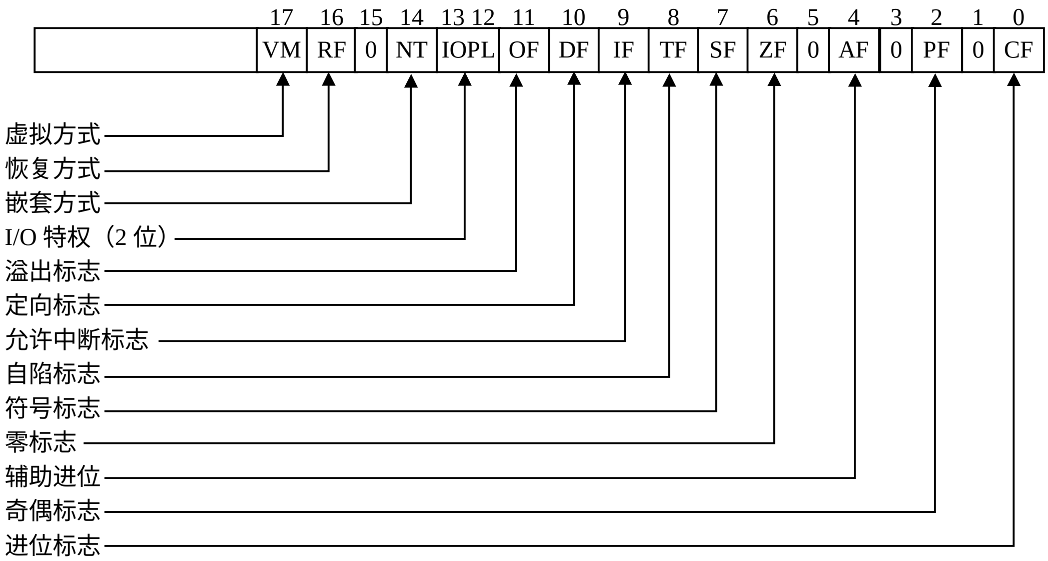


图 2.2　i386 标志寄存器（EFLAGS）

在这些标志位中，我们只介绍在 Linux 内核代码中常用且重要的几个标志位。

第 8 位 TF（Trap Flag）是自陷标志，当将其置 1 时则可以进行单步执行。当指令执行完后，就可能产生异常 1 的自陷（参看第四章）。也就是说，在程序的执行过程中，每执行完一条指令，都要由异常 1 处理程序 （在 Linux 内核中叫做 debug（））进行检验。当将第 8 位清 0 后，且将断点地址装入调试寄存器 DR0～DR3 时，才会产生异常 1 的自陷。

第 12、13 位 IOPL 是输入输出特权级位，这是保护模式下要使用的两个标志位。由于输入输出特权级标志共两位，它的取值范围只可能是 0、1、2 和 3 共 4 个值，恰好与输入输出特权级 0～3 级相对应。但 Linux 内核只使用了两个级别，即 0 和 3 级，0 表示内核级，3 表示用户级。在当前任务的特权级 CPL（Current Privilege Level）高于或等于输入输出特权级时，就可以执行像 IN、OUT、INS、OUTS、STI、CLI 和 LOCK 等指令而不会产生异常 13（即保护异常）。在当前任务特权级 CPL 为 0 时，POPF（从栈中弹出至标志位）指令和中断返回指

令 IRET 可以改变 IOPL 字段的值。

第 9 位 IF（Interrupt Flag）是中断标志位，是用来表示允许或者禁止外部中断（参看第四章）。若第 9 位 IF 被置为 1，则允许 CPU 接收外部中断请求信号；若将 IF 位清 0，则表示禁止外部中断。在保护模式下，只有当第 12、13 位指出当前 CPL 为最高特权级时，才允许将新值置入标志寄存器（EFLAGS）以改变 IF 位的值。

第 10 位 DF（Direction Flag）是定向标志。DF 位规定了在执行串操作的过程中，对源变址寄存器 ESI 或目标变址寄存器 EDI 是增值还是减值。如果 DF 为 1，则寄存器减值；若 DF 为 0，则寄存器值增加。

第 14 位 NT 是嵌套任务标志位。在保护模式下常使用这个标志。当 80386 在发生中断和执行 CALL 指令时就有可能引起任务切换。若是由于中断或由于执行 CALL 指令而出现了任务切换，则将 NT 置为 1。若没有任务切换，则将 NT 位清 0。

第 17 位 VM（Virtual 8086 Mode Flag）是虚拟 8086 方式标志，是 80386 新设置的一个标志位。表示 80386 CPU 是在虚拟 8086 环境中运行。如果 80386 CPU 是在保护模式下运行，而 VM 为又被置成 1，这时 80386 就转换成虚拟 8086 操作方式，使全部段操作就像是在 8086 CPU 上运行一样。VM 位只能由两种方式中的一种方式给予设置，即或者是在保护模式下，由最高特权级（0）级代码段的中断返回指令 IRET 设置，或者是由任务转换进行设置。Linux 内核实现了虚拟 8086 方式，但在本书中我们不准备对此进行详细讨论。

从上面的介绍可以看出，要正确理解标志寄存器（EFLAGS）的各个标志需要很多相关的知识，有些内容在本章的后续部分还会涉及到。在后面的章节中，你会体会会如何灵活应用这些标志。

2．控制寄存器

状态和控制寄存器组除了 EFLAGS、EIP，还有 4 个 32 位的控制寄存器，它们是 CR0，CR1、CR2 和 CR3。现在我们详细看看它们的结构，如图 2.3 所示。

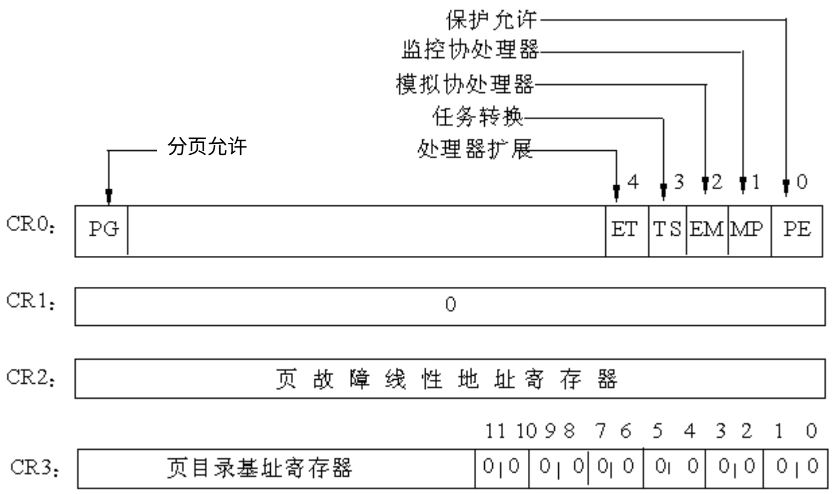


图 2.3　386 中的控制寄存器组

这几个寄存器中保存全局性和任务无关的机器状态。

CR0 中包含了 6 个预定义标志，0 位是保护允许位 PE（Protedted Enable），用于启动保护模式，如果 PE 位置 1，则保护模式启动，如果 PE=0，则在实模式下运行。1 位是监控协处理位 MP（Moniter Coprocessor），它与第 3 位一起决定：当 TS=1 时操作码 WAIT 是否产生一个“协处理器不能使用”的出错信号。第 3 位是任务转换位（Task Switch），当一个任务转换完成之后，自动将它置 1。随着 TS=1，就不能使用协处理器。CR0 的第 2 位是模拟协处理器位 EM （Emulate Coprocessor），如果 EM=1，则不能使用协处理器，如果 EM=0，则允许使用协处理器。第 4 位是微处理器的扩展类型位 ET（Processor Extension Type），其内保存着处理器扩展类型的信息，如果 ET=0，则标识系统使用的是 287 协处理器，如果 ET=1，则表示系统使用的是 387 浮点协处理器。CR0 的第 31 位是分页允许位（Paging Enable），它表示芯片上的分页部件是否允许工作，下一节就会讲到。PG 位和 PE 位定义的操作方式如图 2.4 所示。

PG	PE	方式
0	0	实模式，8080操作
0	1	保护模式，但不允许分页
1	0	出错
1	1	允许分页的保护模式

图 2.4　PG 位和 PE 位定义的操作方式

CR1 是未定义的控制寄存器，供将来的处理器使用。

CR2 是页故障线性地址寄存器，保存最后一次出现页故障的全 32 位线性地址。

CR3 是页目录基址寄存器，保存页目录表的物理地址。页目录表总是放在以 4KB 为单位的存储器边界上，因此，它的地址的低 12 位总为 0，不起作用，即使写上内容，也不会被理会。

这几个寄存器是与分页机制密切相关的，因此，在进程管理及虚拟内存管理中会涉及到这几个寄存器，读者要记住 CR0、CR2 及 CR3 这 3 个寄存器的内容。

2.1.4　系统地址寄存器

80386 有 4 个系统地址寄存器，如图 2.5 所示，它保存操作系统要保护的信息和地址转换表信息。

这 4 个专用寄存器用于引用在保护模式下所需要的表和段，它们的名称和作用如下。

- 全局描述符表寄存器 GDTR（Global Descriptor Table Register ），是 48 位寄存器，用来保存全局描述符表（GDT）的 32 位基地址和 16 位 GDT 的界限。
- 中断描述符表寄存器 IDTR（Interrupt Descriptor Table Register），是 48 位寄存器，用来保存中断描述符表（IDT）的 32 位基地址和 16 位 IDT 的界限。

- 局部描述符表寄存器 LDTR（Global Descriptor Table Register），是 16 位寄存器，保存局部描述符表 LDT 段的选择符。

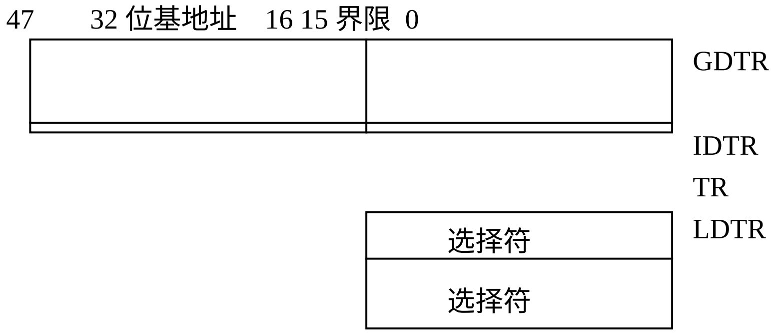


图 2.5　80386 系统地址寄存器

- 任务状态寄存器 TR（Task State Register）是 16 位寄存器，用于保存任务状态段 TSS 段的 16 位选择符。

用以上 4 个寄存器给目前正在执行的任务（或进程）定义任务环境、地址空间和中断向量空间。有关全局描述符表 GST、中断描述符表 IDT、局部描述符表 LDT 及任务状态段 TSS 的具体内容将在稍后进行详细描述。

2.1.5　调试寄存器和测试寄存器

1．调试寄存器

80386 为调试提供了硬件支撑。在 80386 芯片内有 8 个 32 位的调试寄存器 DR0~DR7，如图 2.6 所示。

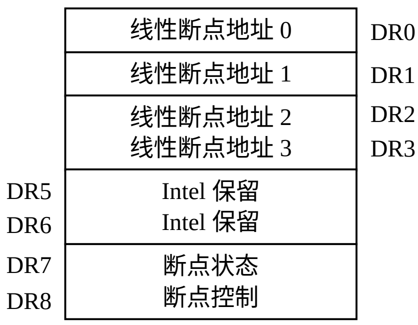


图 2.6　80386 的调试寄存器

这些寄存器可以使系统程序设计人员定义 4 个断点，用它们可以规定指令执行和数据读写的任何组合。DR0~DR3 是线性断点地址寄存器，其中保存着 4 个断点地址。DR4、DR5 是两个备用的调试寄存器，目前尚未定义。DR6 是断点状态寄存器，其低序位是指示符位，

当允许故障调试并检查出故障而进入异常调试处理程序（debug（））时，由硬件把指示符位置 1，调试异常处理程序在退出之前必须把这几位清 0。DR7 是断点控制寄存器，它的高序半个字又被分为 4 个字段，用来规定断点字段的长度是 1 个字节、2 个字节、4 个字节及规定将引起断点的访问类型。低序半个字的位字段用于“允许”断点和“允许”所选择的调试条件。

2．测试寄存器

80386 有两个 32 位的测试寄存器 TR6 和 TR7。这两个寄存器用于在转换旁路缓冲器（Translation Lookaside Buffer）中测试随机存储器（RAM）和相联存储器（CAM）。TR6 是测试命令寄存器，其内存放测试控制命令。TR7 是数据寄存器，其内保存转换旁路缓冲器测试的数据。

2.2　内存地址

在任何一台计算机上，都存在一个程序能产生的内存地址的集合。当程序执行这样一条指令时：

```
MOVE  REG，ADDR
```

它把地址为 ADDR（假设为 10000）的内存单元的内容复制到 REG 中，地址 ADDR 可以通过索引、基址寄存器、段寄存器和其他方式产生。

在 8086 的实模式下，把某一段寄存器左移 4 位，然后与地址 ADDR 相加后被直接送到内存总线上，这个相加后的地址就是内存单元的物理地址，而程序中的这个地址就叫逻辑地址（或叫虚地址）。在 80386 的保护模式下，这个逻辑地址不是被直接送到内存总线，而是被送到内存管理单元（MMU）。MMU 由一个或一组芯片组成，其功能是把逻辑地址映射为物理地址，即进行地址转换，如图 2.7 所示。

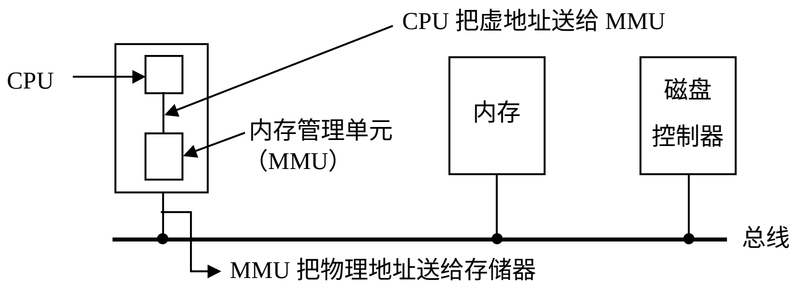


图 2.7　MMU 的位置和功能

当使用 80386 时，我们必须区分以下 3 种不同的地址。

1．逻辑地址

机器语言指令仍用这种地址指定一个操作数的地址或一条指令的地址。这种寻址方式在

Intel 的分段结构中表现得尤为具体，它使得 MS-DOS 或 Windows 程序员把程序分为若干段。每个逻辑地址都由一个段和偏移量组成。

2．线性地址

线性地址是一个 32 位的无符号整数，可以表达高达 2^{32}（4GB）的地址。通常用 16 进制表示线性地址，其取值范围为 0x00000000～0xffffffff。

3．物理地址

物理地址是内存单元的实际地址，用于芯片级内存单元寻址。物理地址也由 32 位无符号整数表示。

从图 2.7 可以看出，MMU 是一种硬件电路，它包含两个部件，一个是分段部件，一个是分页部件，在本书中，我们把它们分别叫做分段机制和分页机制，以利于从逻辑的角度来理解硬件的实现机制。分段机制把一个逻辑地址转换为线性地址；接着，分页机制把一个线性地址转换为物理地址，如图 2.8 所示。

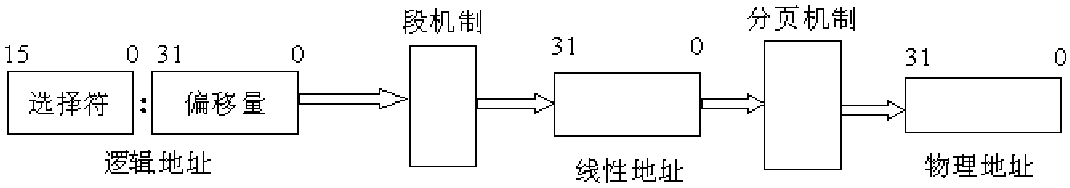


图 2.8　MMU 把逻辑地址转换为物理地址

2.3　段机制和描述符

2.3.1　段机制

在 80386 的段机制中，逻辑地址由两部分组成，即段部分（选择符）及偏移部分。

段是形成逻辑地址到线性地址转换的基础。如果我们把段看成一个对象的话，那么对它的描述如下。

（1）段的基地址（Base Address）：在线性地址空间中段的起始地址。

（2）段的界限（Limit）：表示在逻辑地址中，段内可以使用的最大偏移量。

（3）段的属性（Attribute）： 表示段的特性。例如，该段是否可被读出或写入，或者该段是否作为一个程序来执行，以及段的特权级等。

段的界限定义逻辑地址空间中段的大小。段内在偏移量从 0 到 limit 范围内的逻辑地址，对应于从 Base 到 Base+Limit 范围内的线性地址。在一个段内，偏移量大于段界限的逻辑地址将没有意义，使用这样的逻辑地址，系统将产生异常。另外，如果要对一个段进行访问，

系统会根据段的属性检查访问者是否具有访问权限，如果没有，则产生异常。例如，在 80386 中，如果要在只读段中进行写入，80386 将根据该段的属性检测到这是一种违规操作，则产生异常。

图 2.9 表示一个段如何从逻辑地址空间，重新定位到线性地址空间。图的左侧表示逻辑地址空间，定义了 A、B 及 C 三个段，段容量分别为 $Limit_A$、$Limit_B$ 及 $Limit_C$。图中虚线把逻辑地址空间中的段 A、B 及 C 与线性地址空间区域连接起来表示了这种转换。

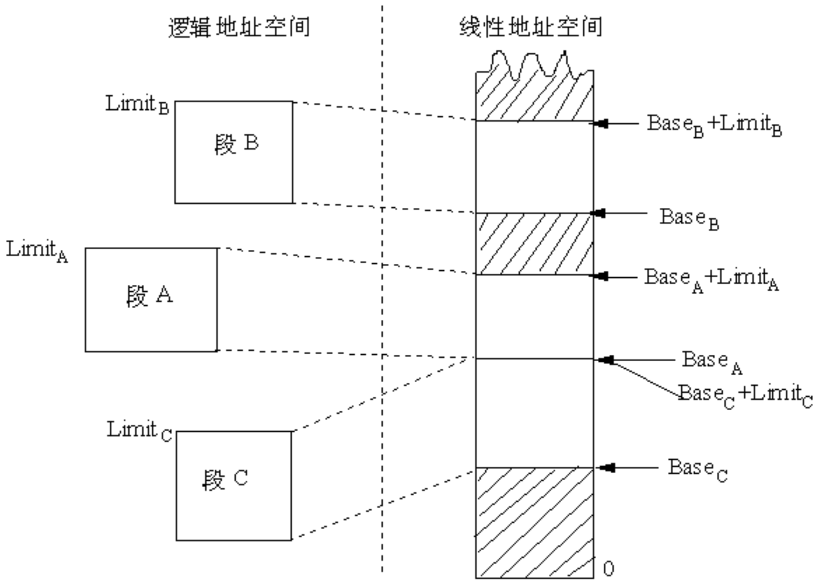


图 2.9 逻辑—线性地址转换

段的基地址、界限及保护属性，存储在段的描述符表中，在逻辑—线性地址转换过程中要对描述符进行访问。段描述符又存储在存储器的段描述符表中，该描述符表是段描述符的一个数组，关于这些内容，我们将在后面详细介绍。

2.3.2 描述符的概念

所谓描述符（Descriptor），就是描述段的属性的一个 8 字节存储单元。在实模式下，段的属性不外乎是代码段、堆栈段、数据段、段的起始地址、段的长度等，而在保护模式下则复杂一些。80386 将它们结合在一起用一个 8 字节的数表示，称为描述符。80386 的一个通用的段描述符的结构如图 2.10 所示。

从图可以看出，一个段描述符指出了段的 32 位基地址和 20 位段界限（即段长）。

第 6 个字节的 G 位是粒度位，当 G=0 时，段长表示段格式的字节长度，即一个段最长可达 1M 字节。当 G=1 时，段长表示段的以 4K 字节为一页的页的数目，即一个段最长可达 1M×4K=4G 字节。D 位表示缺省操作数的大小，如果 D=0，操作数为 16 位，如果 D=1，操作数为 32 位。第 6 个字节的其余两位为 0，这是为了与将来的处理器兼容而必须设置为 0 的位。

字节：	
0	7～0位段界限
1	15～8位段界限
2	7～0位段基址
3	15～8位基址
4	23～16段基址
5	存取权字节
6	G | D | 0 | 0 | 19～16段界限
7	31～24位段基址

图 2.10　段描述符的一般格式

第 5 个字节是存取权字节，它的一般格式如图 2.11 所示。

7	6 5	4	3 2 1	0
P	DPL	S	类　型	A

图 2.11 存取权字节的一般格式

第 7 位 P 位（Present） 是存在位，表示段描述符描述的这个段是否在内存中，如果在内存中。P=1；如果不在内存中，P=0。

DPL（Descriptor Privilege Level），就是描述符特权级，它占两位，其值为 0～3，用来确定这个段的特权级即保护等级。

S 位（System）表示这个段是系统段还是用户段。如果 S=0，则为系统段，如果 S=1，则为用户程序的代码段、数据段或堆栈段。系统段与用户段有很大的不同，后面会具体介绍。

类型占 3 位，第 3 位为 E 位，表示段是否可执行。当 E=0 时，为数据段描述符，这时的第 2 位 ED 表示扩展方向。当 ED=0 时，为向地址增大的方向扩展，这时存取数据段中的数据的偏移量必须小于或等于段界限，当 ED=1 时，表示向地址减少的方向扩展，这时偏移量必须大于界限。当表示数据段时，第 1 位（W）是可写位，当 W=0 时，数据段不能写，W=1 时，数据段可写入。在 80386 中，堆栈段也被看成数据段，因为它本质上就是特殊的数据段。当描述堆栈段时，ED=0，W=1，即堆栈段朝地址增大的方向扩展。

也就是说，当段为数据段时，存取权字节的格式如图 2.12 所示。

当段为代码段时，第 3 位 E=1，这时第 2 位为一致位（C）。当 C=1 时，如果当前特权级低于描述符特权级，并且当前特权级保持不变，那么代码段只能执行。所谓当前特权级（Current Privilege Level），就是当前正在执行的任务的特权级。第 1 位为可读位 R，当 R=0 时，代码段不能读，当 R=1 时可读。也就是说，当段为代码段时，存取权字节的格式如

图 2.13 所示。

7	6 5	4	3	2	1	0
P	DPL	1	0	ED	W	A

图 2-12 数据段的存取字节

7	6 5	4	3	2	1	0
P	DPL	1	1	C	R	A

图 2.13　代码段的存取字节

存取权字节的第 0 位 A 位是访问位，用于请求分段不分页的系统中，每当该段被访问时，将 A 置 1。对于分页系统，则 A 被忽略未用。

2.3.3　系统段描述符

以上介绍了用户段描述符。系统段描述符的一般格式如图 2.14 所示。

字节：					
0	7～0位段界限				
1	15～8位段界限				
2	7～0位段基址				
3	15～8位基址				
4	23～16段基址				
5	P	DPL		0	类　型
6	G	0	0	0	19～16段界限
7	31～24位段基址				

图 2.14　系统段描述符的一般格式

可以看出，系统段描述符的第 5 个字节的第 4 位为 0，说明它是系统段描述符，类型占 4 位，没有 A 位。第 6 个字节的第 6 位为 0，说明系统段的长度是字节粒度，所以，一个系统段的最大长度为 1M 字节。

系统段的类型为 16 种，如图 2.15 所示。

在这 16 种类型中，保留类型和有关 286 的类型不予考虑。

门也是一种描述符，有调用门、任务门、中断门和陷阱门 4 种门描述符。有关门描述符的内容将在第四章中进行具体讨论。

类型号	定义	类型号	定义
0	未定义(Intel公司保留)	8	未定义(Intel公司保留)
1	有效的286TSS	9	有效的386TSS
2	LDT	A	386TSS忙
3	286TSS忙	B	未定义(Intel公司保留)
4	286调用门	C	386调用门
5	任务门	D	未定义(Intel公司保留)
6	286中断门	E	386中断门
7	286陷阱门	F	386陷阱门

图 2.15　系统段的类型

2.3.4　描述符表

各种各样的用户描述符和系统描述符，都放在对应的全局描述符表、局部描述符表和中断描述符表中。

描述符表（即段表）定义了 386 系统的所有段的情况。所有的描述符表本身都占据一个字节为 8 的倍数的存储器空间，空间大小在 8 个字节（至少含一个描述符）到 64K 字节（至多含 8K）个描述符之间。

1．全局描述符表（GDT）

全局描述符表 GDT（Global Descriptor Table），除了任务门，中断门和陷阱门描述符外，包含着系统中所有任务都共用的那些段的描述符。它的第一个 8 字节位置没有使用。

2．中断描述符表（IDT）

中断描述符表 IDT（Interrupt Descriptor Table），包含 256 个门描述符。IDT 中只能包含任务门、中断门和陷阱门描述符，虽然 IDT 表最长也可以为 64K 字节，但只能存取 2K 字节以内的描述符，即 256 个描述符，这个数字是为了和 8086 保持兼容。

3．局部描述符表（LDT）

局部描述符表 LDT（Local Descriptor Table），包含了与一个给定任务有关的描述符，每个任务各自有一个的 LDT。有了 LDT，就可以使给定任务的代码、数据与别的任务相隔离。

每一个任务的局部描述符表 LDT 本身也用一个描述符来表示，称为 LDT 描述符，它包含了有关局部描述符表的信息，被放在全局描述符表 GDT 中。

2.3.5　选择符与描述符表寄存器

在实模式下，段寄存器存储的是真实的段地址，在保护模式下，16 位的段寄存器无法放

下 32 位的段地址，因此，它们被称为选择符，即段寄存器的作用是用来选择描述符。选择符的结构如图 2.16 所示。

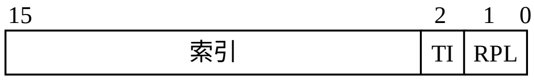


图 2.16　选择符的结构

可以看出，选择符有 3 个域：第 15~3 位这 13 位是索引域，表示的数据为 0~8129，用于指向全局描述符表中相应的描述符。第 2 位为选择域，如果 TI=1，就从局部描述符表中选择相应的描述符，如果 TI=0，就从全局描述符表中选择描述符。第 1、0 位是特权级，表示选择符的特权级，被称为请求者特权级 RPL（Requestor Privilege Level）。只有请求者特权级 RPL 高于（数字低于）或等于相应的描述符特权级 DPL，描述符才能被存取，这就可以实现一定程度的保护。

我们知道，实模式下是直接在段寄存器中放置段基地址，现在则是通过它来存取相应的描述符来获得段基地址和其他信息，这样以来，存取速度会不会变慢呢？为了解决这个问题，386 的每一个段选择符都有一个程序员不可见（也就是说程序员不能直接操纵）的 88 位宽的段描述符高速缓冲寄存器与之对应。无论什么时候改变了段寄存器的内容，只要特权级合理，描述符表中的相应的 8 字节描述符就会自动从描述符表中取出来，装入高速缓冲寄存器中（还有 24 位其他内容）。一旦装入，以后对那个段的访问就都使用高速缓冲寄存器的描述符信息，而不会再重新从表中去取，这就大大加快了执行的时间，如图 2.17 所示。

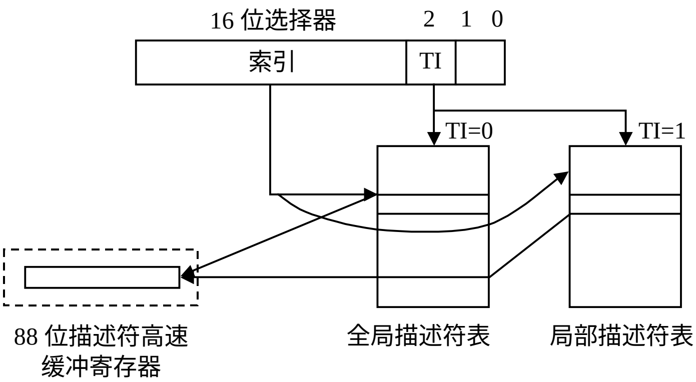


图 2.17　段描述符高速缓冲寄存器的作用

由于段描述符高速缓冲寄存器的内容只有在重新设置选择符时才被重新装入，所以，当你修改了选择符所选择的描述符后，必须对相应的选择符重新装入，这样，88 位描述符高速缓冲寄存器的内容才会发生变化。无论如何，当选择符的值改变时，处理器自动装载不可见部分。

下面讲一下在没有分页操作时，寻址一个存储器操作数的步骤。

（1）在段选择符中装入 16 位数，同时给出 32 位地址偏移量（比如在 ESI、EDI 中等）。

（2）根据段选择符中的索引值、TI 及 RPL 值，再根据相应描述符表寄存器中的段地址和

段界限，进行一系列合法性检查（如特权级检查、界限检查），该段无问题，就取出相应的描述符放入段描述符高速缓冲寄存器中。

(4) 将描述符中的 32 位段基地址和放在 ESI、EDI 等中的 32 位有效地址相加，就形成了 32 位物理地址。

注意：在保护模式下，32 位段基地址不必向左移 4 位，而是直接和偏移量相加形成 32 位物理地址（只要不溢出）。这样做的好处是：段不必再定位在被 16 整除的地址上，也不必左移 4 位再相加。

寻址过程如图 2.18 所示。

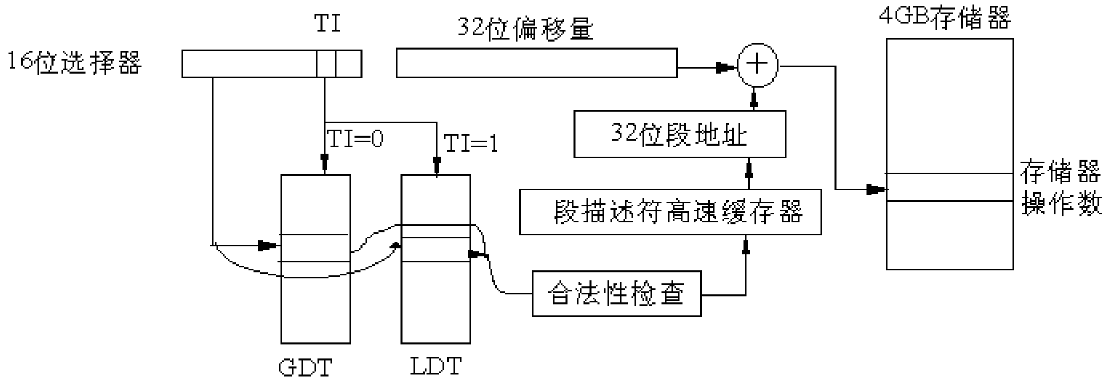


图 2.18　寻址过程

2.3.6　描述符投影寄存器

为了避免在每次存储器访问时，都要访问描述符表，读出描述符并对段进行译码以得到描述符本身的各种信息，每个段寄存器都有与之相联系的描述符投影寄存器。在这些寄存器中，容纳有由段寄存器中的选择符确定的段的描述符信息。段寄存器对编程人员是可见的，而与之相联系的容纳描述符的寄存器，则对编程人员是不可见的，故称之为投影寄存器。图 2.19 中所示的是 6 个寄存器及其投影寄存器。用实线画出的寄存器是段寄存器，用以表示这些寄存器对编程人员可见；用虚线画出的寄存器是投影寄存器，表示对编程人员不可见。

投影寄存器容纳有相应段寄存器寻址的段的基地址、界限及属性。每当用选择符装入段寄存器时，CPU 硬件便自动地把描述符的全部内容装入对应的投影寄存器。因此，在多次访问同一段时，就可以用投影寄存器中的基地址来访问存储器。投影寄存器存储在 80386 的芯片上，因而可以由段基址硬件进行快速访问。因为多数指令访问的数据是在其选择符已经装入到段寄存器之后进行的，所以使用投影寄存器可以得到很好的执行性能。

2.3.7　Linux 中的段

Intel 微处理器的段机制是从 8086 开始提出的， 那时引入的段机制解决了从 CPU 内部 16 位地址到 20 位实地址的转换。为了保持这种兼容性，386 仍然使用段机制，但比以前复杂

得多。因此，Linux 内核的设计并没有全部采用 Intel 所提供的段方案，仅仅有限度地使用了一下分段机制。这不仅简化了 Linux 内核的设计，而且为把 Linux 移植到其他平台创造了条件，因为很多 RISC 处理器并不支持段机制。但是，对段机制相关知识的了解是进入 Linux 内核的必经之路。

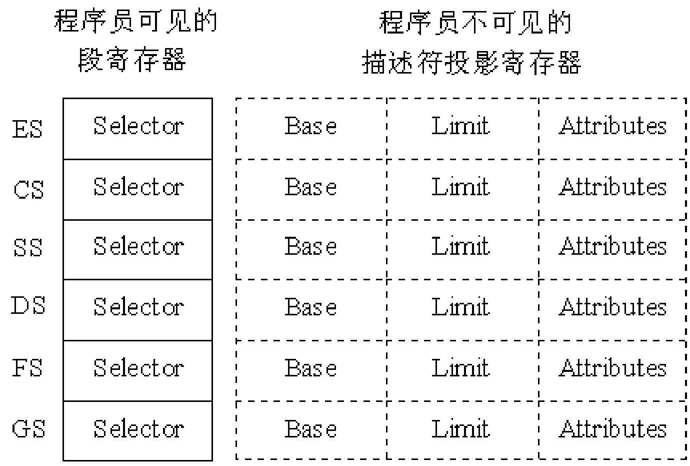


图 2.19 描述符投影寄存器

从 2.2 版开始，Linux 让所有的进程（或叫任务）都使用相同的逻辑地址空间，因此就没有必要使用局部描述符表 LDT。但内核中也用到 LDT，那只是在 VM86 模式中运行 Wine 时，即在 Linux 上模拟运行 Windows 软件或 DOS 软件的程序时才使用。

Linux 在启动的过程中设置了段寄存器的值和全局描述符表 GDT 的内容，段的定义在 include/asm-i386/segment.h 中：

```
#define __KERNEL_CS0x10     ／∗内核代码段，index=2,TI=0,RPL=0∗／
#define __KERNEL_DS0x18     ／∗内核数据段，index=3,TI=0,RPL=0∗／
#define __USER_CS  0x23     ／∗用户代码段，index=4,TI=0,RPL=3∗／
#define __USER_DS  0x2B     ／∗用户数据段，index=5,TI=0,RPL=3∗／
```

从定义看出，没有定义堆栈段，实际上，Linux 内核不区分数据段和堆栈段，这也体现了 Linux 内核尽量减少段的使用。因为没有使用 LDT，因此，TI=0，并把这 4 个段都放在 GDT 中，index 就是某个段在 GDT 表中的下标。内核代码段和数据段具有最高特权，因此其 RPL 为 0，而用户代码段和数据段具有最低特权，因此其 RPL 为 3。可以看出，Linux 内核再次简化了特权级的使用，使用了两个特权级而不是 4 个。

全局描述符表的定义在 arch/i386/kernel/head.S 中：

```
ENTRY（gdt_table)
    .quad 0x0000000000000000   /* NULL descriptor */
    .quad 0x0000000000000000   /* not used */
    .quad 0x00cf9a000000ffff   /* 0x10 kernel 4GB code at 0x00000000 */
    .quad 0x00cf92000000ffff   /* 0x18 kernel 4GB data at 0x00000000 */
    .quad 0x00cffa000000ffff   /* 0x23 user   4GB code at 0x00000000 */
    .quad 0x00cff2000000ffff   /* 0x2b user   4GB data at 0x00000000 */
    .quad 0x0000000000000000   /* not used */
```

```
    .quad 0x0000000000000000    /* not used */
    /*
     * The APM segments have byte granularity and their bases
     * and limits are set at run time.
    */
    .quad 0x0040920000000000    /* 0x40 APM set up for bad BIOS's */
    .quad 0x00409a0000000000    /* 0x48 APM CS    code */
    .quad 0x00009a0000000000    /* 0x50 APM CS 16 code  (16 bit)  */
    .quad 0x0040920000000000    /* 0x58 APM DS    data */
    .fill NR_CPUS*4,8,0   /* space for TSS's and LDT's */
```

从代码可以看出，GDT 放在数组变量 gdt_table 中。按 Intel 规定，GDT 中的第一项为空，这是为了防止加电后段寄存器未经初始化就进入保护模式而使用 GDT 的。第二项也没用。从下标 2～5 共 4 项对应于前面的 4 种段描述符值。对照图 2.10，从描述符的数值可以得出：

- 段的基地址全部为 0x00000000；
- 段的上限全部为 0xffff；
- 段的粒度 G 为 1，即段长单位为 4KB；
- 段的 D 位为 1，即对这 4 个段的访问都为 32 位指令；
- 段的 P 位为 1，即 4 个段都在内存。

由此可以得出，每个段的逻辑地址空间范围为 0～4GB。读者可能对此不太理解，但只要对照图 2.9 就可以发现，这种设置既简单又巧妙。因为每个段的基地址为 0，因此，逻辑地址到线性地址映射保持不变，也就是说，偏移量就是线性地址，我们以后所提到的逻辑地址（或虚拟地址）和线性地址指的也就是同一地址。看来，Linux 巧妙地把段机制给绕过去了，而完全利用了分页机制。

从逻辑上说，Linux 巧妙地绕过了逻辑地址到线性地址的映射，但实质上还得应付 Intel 所提供的段机制。只不过，Linux 把段机制变得相当简单，它只把段分为两种：用户态（RPL＝3）的段和内核态（RPL=0）的段，因此，描述符投影寄存器的内容很少发生变化，只在进程从用户态切换到内核态或者反之时才发生变化。另外，用户段和内核段的区别也仅仅在其 RPL 不同，因此内核根本无需访问描述符投影寄存器，当然也无需访问 GDT，而仅从段寄存器的最低两位就可以获取 RPL 的信息。Linux 这样设计所带来的好处是显而易见的，Intel 的分段部件对 Linux 性能造成的影响可以忽略不计。

在上面描述的 GDT 表中，紧接着那 4 个段描述的两个描述符被保留，然后是 4 个高级电源管理（APM）特征描述符，对此不进行详细讨论。

按 Intel 的规定，每个进程有一个任务状态段（TSS）和局部描述符表 LDT，但 Linux 也没有完全遵循 Intel 的设计思路。如前所述，Linux 的进程没有使用 LDT，而对 TSS 的使用也非常有限，每个 CPU 仅使用一个 TSS。

通过上面的介绍可以看出，Intel 的设计可谓周全细致，但 Linux 的设计者并没有完全陷入这种沼泽，而是选择了简洁而有效的途径，以完成所需功能并达到较好的性能为目标。

2.4　分页机制

分页机制在段机制之后进行，以完成线性—物理地址的转换过程。段机制把逻辑地址转换为线性地址，分页机制进一步把该线性地址再转换为物理地址。

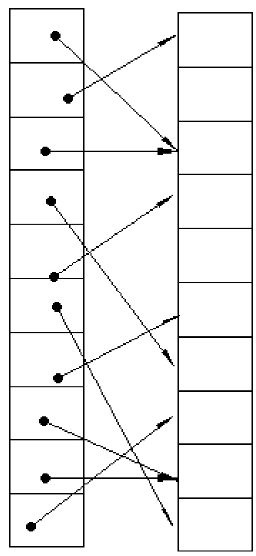


图 2.20　分页机制把线性地址转换为物理地址

分页机制由 CR0 中的 PG 位启用。如 PG=1，启用分页机制，并使用本节要描述的机制，把线性地址转换为物理地址。如 PG=0，禁用分页机制，直接把段机制产生的线性地址当作物理地址使用。分页机制管理的对象是固定大小的存储块，称之为页（page）。分页机制把整个线性地址空间及整个物理地址空间都看成由页组成，在线性地址空间中的任何一页，可以映射为物理地址空间中的任何一页（我们把物理空间中的一页叫做一个页面或页框（page frame））。图 2.20 所示出分页机制如何把线性地址空间及物理地址空间划分为页，以及如何在这两个地址空间进行映射。图 2.20 的左边是线性地址空间，并将其视为一个具有一页大小的固定块的序列。图 2.20 的右边是物理地址空间，也将其视为一个页面的序列。图中，用箭头把线性地址空间中的页，与对应的物理地址空间中的页面联系起来。在这里，线性地址空间中的页与物理地址空间中的页是随意地对应起来的。

80386 使用 4K 字节大小的页。每一页都有 4K 字节长，并在 4K 字节的边界上对齐，即每一页的起始地址都能被 4K 整除。因此，80386 把 4G 字节的线性地址空间，划分为 1G 个页面，每页有 4K 字节大小。分页机制通过把线性地址空间中的页，重新定位到物理地址空间来进行管理，因为每个页面的整个 4K 字节作为一个单位进行映射，并且每个页面都对齐 4K 字节的边界，因此，线性地址的低 12 位经过分页机制直接地作为物理地址的低 12 位使用。

线性—物理地址的转换，可将其意义扩展为允许将一个线性地址标记为无效，而不是实际地产生一个物理地址。有两种情况可能使页被标记为无效：其一是线性地址是操作系统不

支持的地址；其二是在虚拟存储器系统中，线性地址对应的页存储在磁盘上，而不是存储在物理存储器中。在前一种情况下，程序因产生了无效地址而必须被终止。对于后一种情况，该无效的地址实际上是请求操作系统的虚拟存储管理系统，把存放在磁盘上的页传送到物理存储器中，使该页能被程序所访问。由于无效页通常是与虚拟存储系统相联系的，这样的无效页通常称为未驻留页，并且用页表属性位中叫做存在位的属性位进行标识。未驻留页是程序可访问的页，但它不在主存储器中。对这样的页进行访问，形式上是发生异常，实际上是通过异常进行缺页处理。

2.4.1 分页机构

如前所述，分页是将程序分成若干相同大小的页，每页 4K 个字节。如果不允许分页（CR0 的最高位置 0），那么经过段机制转化而来的 32 位线性地址就是物理地址。但如果允许分页（CR0 的最高位置 1），就要将 32 位线性地址通过一个两级表格结构转化成物理地址。

1．两级页表结构

为什么采用两级页表结构呢？

在 80386 中页表共含 1M 个表项，每个表项占 4 个字节。如果把所有的页表项存储在一个表中，则该表最大将占 4M 字节连续的物理存储空间。为避免使页表占有如此巨额的物理存储器资源，故对页表采用了两级表的结构，而且对线性地址的高 20 位的线性—物理地址转化也分为两部完成，每一步各使用其中的 10 位。

两级表结构的第一级称为页目录，存储在一个 4K 字节的页面中。页目录表共有 1K 个表项，每个表项为 4 个字节，并指向第二级表。线性地址的最高 10 位（即位 31～位 22）用来产生第一级的索引，由索引得到的表项中，指定并选择了 1K 个二级表中的一个表。

两级表结构的第二级称为页表，也刚好存储在一个 4K 字节的页面中，包含 1K 个字节的表项，每个表项包含一个页的物理基地址。第二级页表由线性地址的中间 10 位（即位 21~位 12）进行索引，以获得包含页的物理地址的页表项，这个物理地址的高 20 位与线性地址的低 12 位形成了最后的物理地址，也就是页转化过程输出的物理地址，具体转化过程稍后会讲到，如图 2.21 为两级页表结构。

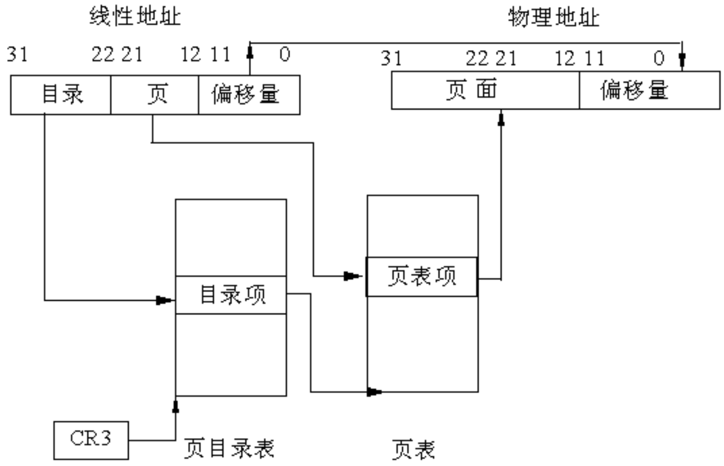


图 2.21　两级页表结构

2．页目录项

图 2-22 所示为页目录表，最多可包含 1024 个页目录项，每个页目录项为 4 个字节，结构如图 2.22 所示。

- 第 31~12 位是 20 位页表地址，由于页表地址的低 12 位总为 0，所以用高 20 位指出 32 位页表地址就可以了。因此，一个页目录最多包含 1024 个页表地址。
- 第 0 位是存在位，如果 P=1，表示页表地址指向的该页在内存中，如果 P=0，表示不在内存中。
- 第 1 位是读/写位，第 2 位是用户/管理员位，这两位为页目录项提供硬件保护。当特权级为 3 的进程要想访问页面时，需要通过页保护检查，而特权级为 0 的进程就可以绕过页保护，如图 2.23 所示。
- 第 3 位是 PWT（Page Write-Through）位，表示是否采用写透方式，写透方式就是既写内存（RAM）也写高速缓存,该位为 1 表示采用写透方式。
- 第 4 位是 PCD（Page Cache Disable）位，表示是否启用高速缓存，该位为 1 表示启用高速缓存。

	7	6	5	4	3	2	1	0
7~0位	PSE	0	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页表地址				OS专用			0
23~16位	11~4位页表地址							
31~24位	19~12位页表地址							

图 2.22　页目录中的页目录项

U/S	R/W	允许级别3	允许级别0
0	0	无	读/写
0	1	无	读/写
1	0	只读	读/写
1	1	读/写	读/写

图 2.23　由 U/S 和 R/W 提供的保护

- 第 5 位是访问位，当对页目录项进行访问时，A 位=1。
- 第 7 位是 Page Size 标志，只适用于页目录项。如果置为 1，页目录项指的是 4MB 的页面，请看后面的扩展分页。
- 第 9~11 位由操作系统专用，Linux 也没有做特殊之用。

2．页面项

80386 的每个页目录项指向一个页表，页表最多含有 1024 个页面项，每项 4 个字节，包含页面的起始地址和有关该页面的信息。页面的起始地址也是 4K 的整数倍，所以页面的低 12 位也留作它用，如图 2.24 所示。

	7	6	5	4	3	2	1	0
7~0位	0	D	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页面地址				OS专用			0
23~16位	11~4位页面地址							
31~24位	19~12位页面地址							

图 2.24　页表中的页面项

第 31～12 位是 20 位物理页面地址，除第 6 位外第 0～5 位及 9～11 位的用途和页目录项一样，第 6 位是页面项独有的，当对涉及的页面进行写操作时，D 位被置 1。

4GB 的存储器只有一个页目录，它最多有 1024 个页目录项，每个页目录项又含有 1024 个页面项，因此，存储器一共可以分成 1024×1024=1M 个页面。由于每个页面为 4K 个字节，所以，存储器的大小正好最多为 4GB。

3．线性地址到物理地址的转换

当访问一个操作单元时，如何由分段结构确定的 32 位线性地址通过分页操作转化成 32 位物理地址呢？过程如图 2.25 所示。

第一步，CR3 包含着页目录的起始地址，用 32 位线性地址的最高 10 位 A31~A22 作为页目录的页目录项的索引，将它乘以 4，与 CR3 中的页目录的起始地址相加，形成相应页表的地址。

第二步，从指定的地址中取出 32 位页目录项，它的低 12 位为 0，这 32 位是页表的起始地址。用 32 位线性地址中的 A21~A12 位作为页表中的页面的索引，将它乘以 4，与页表的起始地址相加，形成 32 位页面地址。

第三步，将 A11~A0 作为相对于页面地址的偏移量，与 32 位页面地址相加，形成 32 位物理地址。

4．扩展分页

从奔腾处理器开始，Intel 微处理器引进了扩展分页，它允许页的大小为 4MB，如图 2.26 所示。

在扩展分页的情况下，分页机制把 32 位线性地址分成两个域：最高 10 位的目录域和其余 22 位的偏移量。

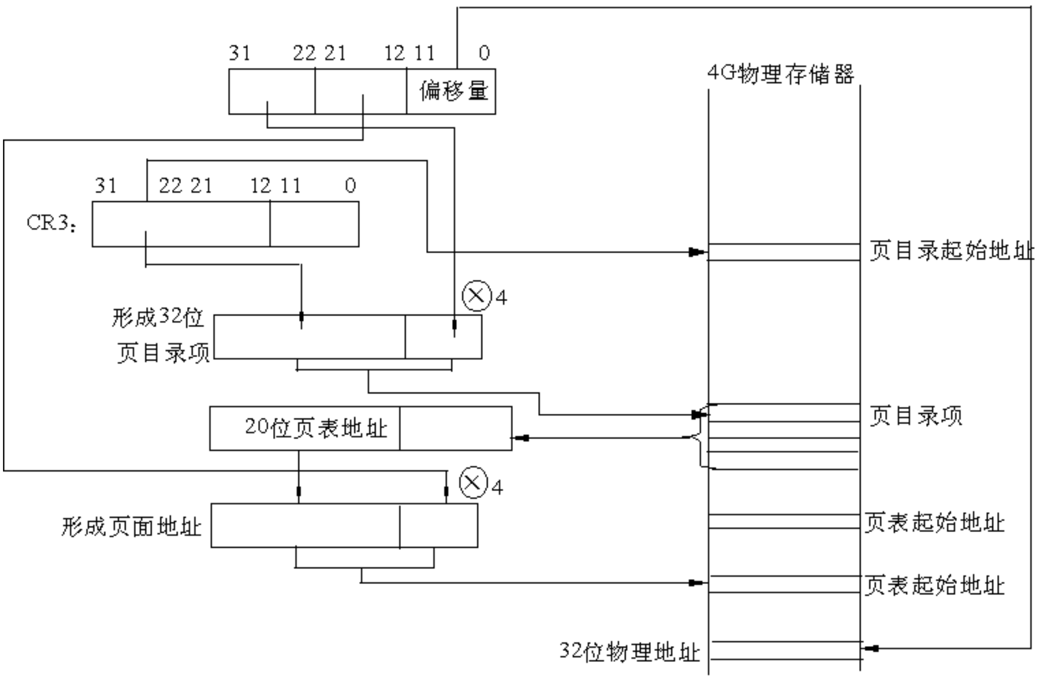


图 2.25　32 位线性地址到物理地址的转换

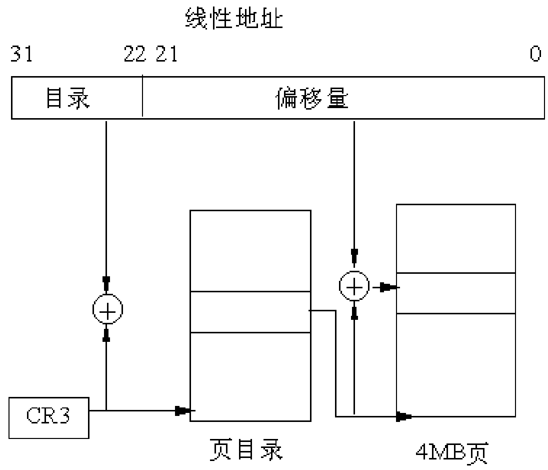


图 2.26　扩展分页

2.4.2　页面高速缓存

由于在分页情况下，每次存储器访问都要存取两级页表，这就大大降低了访问速度。所以，为了提高速度，在 386 中设置一个最近存取页面的高速缓存硬件机制，它自动保持 32 项处理器最近使用的页面地址，因此，可以覆盖 128K 字节的存储器地址。当进行存储器访问时，先检查要访问的页面是否在高速缓存中，如果在，就不必经过两级访问了，如果不在，再进行两级访问。平均来说，页面高速缓存大约有 98%的命中率，也就是说每次访问存储器时，只有 2%的情况必须访问两级分页机构。这就大大加快了速度，页面高速缓存的作用如图 2.27 所示。有些书上也把页面高速缓存叫做“ 联想存储器 ”或“ 转换旁路缓冲器（TLB）”。

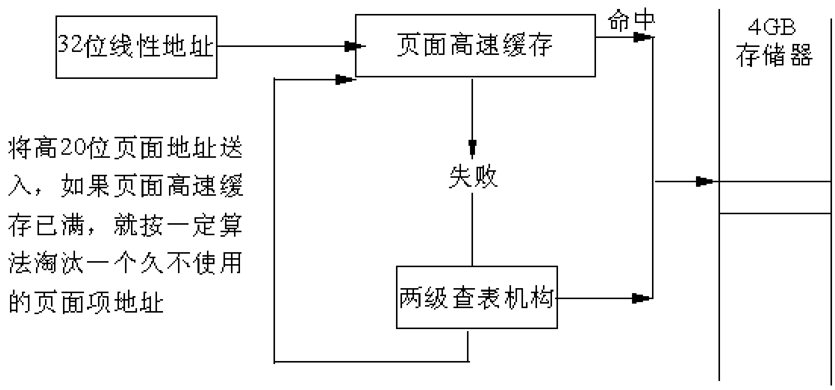


图 2.27　子页面高速缓存

2.5　Linux 中的分页机制

如前所述，Linux 主要采用分页机制来实现虚拟存储器管理，原因如下。

- Linux 的分段机制使得所有的进程都使用相同的段寄存器值，这就使得内存管理变得简单，也就是说，所有的进程都使用同样的线性地址空间（0～4GB）。
- Linux 设计目标之一就是能够把自己移植到绝大多数流行的处理器平台。但是，许多 RISC 处理器支持的段功能非常有限。

为了保持可移植性，Linux 采用三级分页模式而不是两级，这是因为许多处理器（如康柏的 Alpha，Sun 的 UltraSPARC，Intel 的 Itanium）都采用 64 位结构的处理器，在这种情况下，两级分页就不适合了，必须采用三级分页。如图 2.28 所示为三级分页模式，为此，Linux 定义了 3 种类型的页表。

- 总目录 PGD（Page Global Directory）
- 中间目录 PMD（Page Middle Derectory）
- 页表 PT（Page Table）

尽管 Linux 采用的是三级分页模式，但我们的讨论还是以 Intel 奔腾处理器的两级分页模式为主，因此，Linux 忽略中间目录层，以后，我们把总目录就叫页目录。

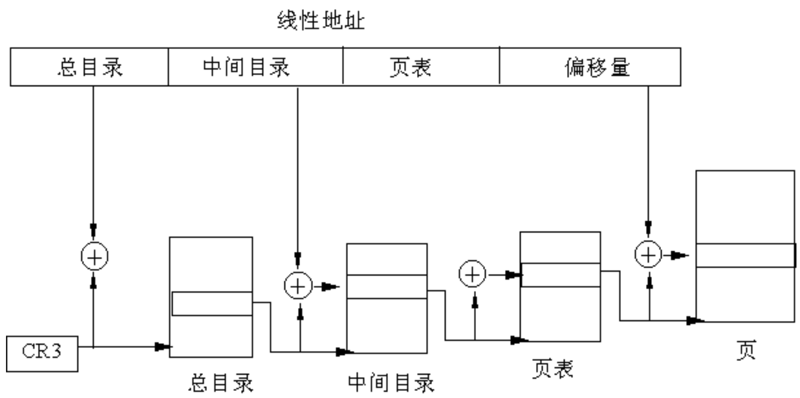


图 2.28　Linux 的三级分页

2.5.1　与页相关的数据结构及宏的定义

上一节讨论的分页机制是硬件对分页的支持，这是虚拟内存管理的硬件基础。要想使这种硬件机制充分发挥其功能，必须有相应软件的支持，我们来看一下 Linux 所定义的一些主要数据结构，其分布在 include/asm-i386/目录下的 page.h，pgtable.h 及 pgtable-2level.h 三个文件中。

1．表项的定义

如上所述，PGD、PMD 及 PT 表的表项都占 4 个字节，因此，把它们定义为无符号长整数，

分别叫做 pgd_t、pmd_t 及 pte_t（pte 即 Page table Entry），在 page.h 中定义如下：

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
```

可以看出，Linux 没有把这几个类型直接定义长整数而是定义为一个结构，这是为了让 gcc 在编译时进行更严格的类型检查。另外，还定义了几个宏来访问这些结构的成分，这也是一种面向对象思想的体现：

```
#define pte_val(x)   ((x).pte_low)
#define pmd_val(x)  ((x).pmd)
#define pgd_val(x)   ((x).pgd)
```

从图 2.22 和图 2.24 可以看出，对这些表项应该定义成位段，但内核并没有这样定义，而是定义了一个页面保护结构 pgprot_t 和一些宏：

```
typedef struct { unsigned long pgprot; } pgprot_t;
#define pgprot_val(x)     ((x).pgprot)
```

字段 pgprot 的值与图 2.24 页面项的低 12 位相对应，其中的 9 位对应 0～9 位，在 pgtalbe.h 中定义了对应的宏：

```
#define _PAGE_PRESENT   0x001
#define _PAGE_RW    0x002
#define _PAGE_USER 0x004
#define _PAGE_PWT   0x008
#define _PAGE_PCD   0x010
#define _PAGE_ACCESSED  0x020
#define _PAGE_DIRTY0x040
#define _PAGE_PSE   0x080   /* 4 MB  (or 2MB)  page, Pentium+, if present.. */
#define _PAGE_GLOBAL     0x100   /* Global TLB entry PPro+ */
```

在你阅读源代码的过程中你能体会到，把标志位定义为宏而不是位段更有利于编码。

另外，页目录表及页表在 pgtable.h 中定义如下：

```
extern pgd_t swapper_pg_dir[1024];
extern unsigned long pg0[1024];
```

swapper_pg_dir 为页目录表，pg0 为一临时页表，每个表最多都有 1024 项。

2．线性地址域的定义

Intel 线性地址的结构如图 2.29 所示。

31　　　　22	21　　　　12	11　　　　0
目录	页	偏移量

图 2.29　32 位的线性地址结构

偏移量的位数

```
#define PAGE_SHIFT 12
#define PAGE_SIZE   (1UL << PAGE_SHIFT)
#define PTRS_PER_PTE     1024
#define PAGE_MASK   (~(PAGE_SIZE-1))
```

其中 PAGE_SHIFT 宏定义了偏移量的位数为 12,因此页大小 PAGE_SIZE 为 2^{12}＝4096 字节；

PTRS_PER_PTE 为页表的项数；最后 PAGE_MASK 值定义为 0xfffff000，用以屏蔽掉偏移量域的所有位（12 位）。

PGDIR_SHIFT

```
#define PGDIR_SHIFT22
#define PTRS_PER_PGD    1024
#define PGDIR_SIZE  (1UL << PGDIR_SHIFT)
#define PGDIR_MASK  (~ (PGDIR_SIZE-1) )
```

PGDIR_SHIFT 是页表所能映射区域线性地址的位数，它的值为 22（12 位的偏移量加上 10 位的页表）；PTRS_PER_PGD 为页目录目录项数；PGDIR_SIZE 为页目录的大小，为 2^{22}，即 4MB；PGDIR_MASK 为 0xffc00000，用于屏蔽偏移量位与页表域的所有位。

（3）PMD_SHIFT

```
#define PMD_SHIFT  22
#define PTRS_PER_PMD    1
```

PMD_SHIFT 为中间目录表映射的地址位数，其值也为 22，但是因为 Linux 在 386 中只用了两级页表结构，因此，让其目录项个数为 1，这就使得中间目录在指针序列中的位置被保存，以便同样的代码在 32 位系统和 64 位系统下都能使用。后面的讨论我们不再提及中间目录。

2.5.2 对页目录及页表的处理

在 page.h，pgtable.h 及 pgtable-2level.h3 个文件中还定义有大量的宏，用以对页目录、页表及表项的处理，我们在此介绍一些主要的宏和函数。

1．表项值的确定

```
static inline int pgd_none (pgd_t pgd)  { return 0; }
static inline int pgd_present (pgd_t pgd)    { return 1; }
#define pte_present (x)    ( (x) .pte_low &  (_PAGE_PRESENT | _PAGE_PROTNONE) )
```

pgd_none（）函数直接返回 0，表示尚未为这个页目录建立映射，所以页目录项为空。pgd_present（）函数直接返回 1，表示映射虽然还没有建立，但页目录所映射的页表肯定存在于内存（即页表必须一直在内存）。

pte_present 宏的值为 1 或 0，表示 P 标志位。如果页表项不为 0，但标志位为 0，则表示映射已经建立，但所映射的物理页面不在内存。

2．清相应表的表项

```
#define pgd_clear (xp)   do { } while  (0)
#define pte_clear (xp)    do { set_pte (xp, __pte (0) ) ; } while  (0)
```

pgd_clear 宏实际上什么也不做，定义它可能是为了保持编程风格的一致。pte_clear 就是把 0 写到页表表项中。

3．对页表表项标志值进行操作的宏

这些宏的代码在 pgtable.h 文件中，表 2.1 给出宏名及其功能。

表 2.1　　　　　　　　　　对页表表项标志值进行操作的宏及其功能

宏名	功能
Set_pte ()	把一个具体的值写入表项
Pte_read ()	返回 User/Supervisor 标志值（由此可以得知是否可以在用户态下访问此页）
Pte _write ()	如果 Present 标志和 Read/Write 标志都为 1，则返回 1（此页是否存在并可写）
Pte _exec ()	返回 User/Supervisor 标志值
Pte _dirty ()	返回 Dirty 标志的值（说明此页是否被修改过）
Pte _young ()	返回 Accessed 标志的值（说明此页是否被存取过）
Pte _wrprotect ()	清除 Read/Write 标志
Pte _rdprotect ()	清除 User/Supervisor 标志
Pte _mkwrite	设置 Read/Write 标志
Pte _mkread	设置 User/Supervisor 标志
Pte _mkdirty ()	把 Dirty 标志置 1
Pte _mkclean ()	把 Dirty 标志置 0
Pte _mkyoung	把 Accessed 标志置 1
Pte _mkold ()	把 Accessed 标志置 0
Pte _modify (p,v)	把页表表项 p 的所有存取权限设置为指定的值 v
Mk_pte ()	把一个线性地址和一组存取权限合并来创建一个 32 位的页表表项
Pte _pte_phys ()	把一个物理地址与存取权限合并来创建一个页表表项
Pte _page ()	从页表表项返回页的线性地址

实际上页表的处理是一个复杂的过程，在这里我们仅仅让读者对软硬件如何结合起来有一个初步的认识，有关页表更多的内容我们将在第六章接着讨论。

2.6　Linux 中的汇编语言

在阅读 Linux 源代码时，你可能碰到一些汇编语言片段，有些汇编语言出现在以.S 为扩展名的汇编文件中，在这种文件中，整个程序全部由汇编语言组成。有些汇编命令出现在以.c 为扩展名的 C 文件中，在这种文件中，既有 C 语言，也有汇编语言，我们把出现在 C 代码中的汇编语言叫所“嵌入式”汇编。不管这些汇编代码出现在哪里，它在一定程度上都成为阅读源代码的拦路虎。

尽管 C 语言已经成为编写操作系统的主要语言，但是，在操作系统与硬件打交道的过程中，在需要频繁调用的函数中以及某些特殊的场合中，C 语言显得力不从心，这时，繁琐但又高效的汇编语言就粉墨登场了。因此，在了解一些硬件的基础上，必须对相关的汇编语言知识也有所了解。

读者可能有过在 DOS 操作系统下编写汇编程序的经历，也具备一定的汇编知识。但是，在 Linux 的源代码中，你可能看到了与 Intel 的汇编语言格式不一样的形式，这就是 AT&T 的 386 汇编语言。

2.6.1　AT&T 与 Intel 汇编语言的比较

我们知道，Linux 是 UNIX 家族的一员，尽管 Linux 的历史不长，但与其相关的很多事情都发源于 UNIX。就 Linux 所使用的 386 汇编语言而言，它也是起源于 UNIX。UNIX 最初是为 PDP-11 开发的，曾先后被移植到 VAX 及 68000 系列的处理器上，这些处理器上的汇编语言都采用的是 AT&T 的指令格式。当 UNIX 被移植到 i386 时，自然也就采用了 AT&T 的汇编语言格式，而不是 Intel 的格式。尽管这两种汇编语言在语法上有一定的差异，但所基于的硬件知识是相同的，因此，如果你非常熟悉 Intel 的语法格式，那么你也可以很容易地把它“移植”到 AT&T。下面我们通过对照 Intel 与 AT&T 的语法格式，以便于你把过去的知识能很快地“移植”过来。

1．前缀

在 Intel 的语法中，寄存器和和立即数都没有前缀。但是在 AT&T 中，寄存器前冠以“%”，而立即数前冠以“$”。在 Intel 的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在 AT&T 中，十六进制立即数前冠以“0x”，如表 2.2 所示给出几个相应的例子。

2．操作数的方向

Intel 与 AT&T 操作数的方向正好相反。在 Intel 语法中，第一个操作数是目的操作数，第二个操作数是源操作数。而在 AT&T 中，第一个数是源操作数，第二个数是目的操作数。由此可以看出，AT&T 的语法符合人们通常的阅读习惯。

例如：在 Intel 中，mov　　eax,[ecx]

在 AT&T 中，movl　(%ecx),%eax

表 2.2　　Intel 与 AT&T 前缀的区别

Intel 语法	AT&T 语法
mov　eax,8	movl $8,%eax
mov　ebx,0ffffh	movl $0xffff,%ebx
int　80h	int　$0x80

3．内存单元操作数

从上面的例子可以看出，内存操作数也有所不同。在 Intel 的语法中，基寄存器用“[]”括起来，而在 AT&T 中，用“()”括起来。

例如：　在 Intel 中，mov　eax,[ebx+5]

在 AT&T，movl　5(%ebx),%eax

4．间接寻址方式

与 Intel 的语法比较，AT&T 间接寻址方式可能更晦涩难懂一些。Intel 的指令格式是 segreg:[base+index*scale+disp]，而 AT&T 的格式是%segreg:disp(base,index,scale)。

其中 index/scale/disp/segreg 全部是可选的，完全可以简化掉。如果没有指定 scale 而指定了 index，则 scale 的缺省值为 1。segreg 段寄存器依赖于指令以及应用程序是运行在实模式还是保护模式下，在实模式下，它依赖于指令，而在保护模式下，segreg 是多余的。在 AT&T 中，当立即数用在 scale/disp 中时，不应当在其前冠以"$"前缀，表 2.3 给出其语法及几个相应的例子。

表 2.3　内存操作数的语法及举例

Intel 语法	AT&T 语法
指令 foo,segreg:[base+index*scale+disp]	指令 %segreg:disp（base,index,scale）,foo
mov　eax,[ebx+20h]	Movl　0x20（%ebx）,%eax
add　eax,[ebx+ecx*2h	Addl　（%ebx,%ecx,0x2）,%eax
lea　eax,[ebx+ecx]	Leal　（%ebx,%ecx）,%eax
sub　eax,[ebx+ecx*4h-20h]	Subl　-0x20（%ebx,%ecx,0x4）,%eax

从表中可以看出，AT&T 的语法比较晦涩难懂，因为[base+index*scale+disp]一眼就可以看出其含义，而 disp（base,index,scale）则不可能做到这点。

这种寻址方式常常用在访问数据结构数组中某个特定元素内的一个字段，其中，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素还是一个结构，则 disp 为具体字段在结构中的位移。

5．操作码的后缀

在上面的例子中你可能已注意到，在 AT&T 的操作码后面有一个后缀，其含义就是指出操作码的大小。"l"表示长整数（32 位），"w"表示字（16 位），"b"表示字节（8 位）。而在 Intel 的语法中，则要在内存单元操作数的前面加上 byte ptr、word ptr 和 dword ptr，"dword"对应"long"。表 2.4 给出了几个相应的例子。

表 2.4　操作码的后缀举例

Intel 语法	AT&T 语法
Mov al,bl	movb　%bl,%al
Mov ax,bx	movw　%bx,%ax
Mov eax,ebx	movl　%ebx,%eax
Mov eax, dword ptr [ebx]	movl　（%ebx）,%eax

2.6.2　AT&T 汇编语言的相关知识

在 Linux 源代码中，以.S 为扩展名的文件是"纯"汇编语言的文件。这里，我们结合具

体的例子再介绍一些 AT&T 汇编语言的相关知识。

1．GNU 汇编程序 GAS（GNU Assembly）和连接程序

当你编写了一个程序后，就需要对其进行汇编（assembly）和连接。在 Linux 下有两种方式，一种是使用汇编程序 GAS 和连接程序 ld，一种是使用 gcc。我们先来看一下 GAS 和 ld:

GAS 把汇编语言源文件（.o）转换为目标文件（.o），其基本语法如下：

```
as filename.s -o filename.o
```

一旦创建了一个目标文件，就需要把它连接并执行，连接一个目标文件的基本语法为：

```
ld filename.o -o filename
```

这里 filename.o 是目标文件名，而 filename 是输出（可执行） 文件。

GAS 使用的是 AT&T 的语法而不是 Intel 的语法，这就再次说明了 AT&T 语法是 UNIX 世界的标准，你必须熟悉它。

如果要使用 GNC 的 C 编译器 gcc，就可以一步完成汇编和连接，例如：

```
gcc -o example example.S
```

这里，example.S 是你的汇编程序，输出文件（可执行文件）名为 example。其中，扩展名必须为大写的 S，这是因为，大写的 S 可以使 gcc 自动识别汇编程序中的 C 预处理命令，像#include、#define、#ifdef、#endif 等，也就是说，使用 gcc 进行编译，你可以在汇编程序中使用 C 的预处理命令。

2．AT&T 中的节（Section）

在 AT&T 的语法中，一个节由.section 关键词来标识，当你编写汇编语言程序时，至少需要有以下 3 种节。

section .data：这种节包含程序已初始化的数据，也就是说，包含具有初值的那些变量，例如：

```
    hello: .string "Hello world!\n"
    hello_len : .long 13
```

.section .bss：这个节包含程序还未初始化的数据，也就是说，包含没有初值的那些变量。当操作系统装入这个程序时将把这些变量都置为 0，例如：

```
name : .fill 30    # 用来请求用户输入名字
name_len : .long  0    # 名字的长度（尚未定义）
```

当这个程序被装入时，name 和 name_len 都被置为 0。如果你在.bss 节不小心给一个变量赋了初值，这个值也会丢失，并且变量的值仍为 0。

使用.bss 比使用.data 的优势在于，.bss 节不占用磁盘的空间。在磁盘上，一个长整数就足以存放.bss 节。当程序被装入到内存时，操作系统也只分配给这个节 4 个字节的内存大小。

注意，编译程序把.data 和.bss 在 4 字节上对齐（align），例如，.data 总共有 34 字节，那么编译程序把它对齐在 36 字节上，也就是说，实际给它 36 字节的空间。

section .text ：这个节包含程序的代码，它是只读节，而.data 和.bss 是读／写节。

3．汇编程序指令（Assembler Directive）

上面介绍的.section 就是汇编程序指令的一种，GNU 汇编程序提供了很多这样的指令（directiv），这种指令都是以句点（.）为开头，后跟指令名（小写字母），在此，我们只介绍在内核源代码中出现的几个指令（以 arch/i386/kernel/head.S 中的代码为例）。

（1）.ascii "string"...

.ascii 表示零个或多个（用逗号隔开）字符串，并把每个字符串（结尾不自动加“0“字节）中的字符放在连续的地址单元。

还有一个与.ascii 类似的.asciz，z 代表“0”，即每个字符串结尾自动加一个“0”字节，例如：

```
int_msg:
    .asciz "Unknown interrupt\n"
```

（2）.byte 表达式

.byte 表示零或多个表达式（用逗号隔开），每个表达式被放在下一个字节单元。

（3）.fill 表达式

形式：.fill repeat , size , value

其中，repeat、size 和 value 都是常量表达式。Fill 的含义是反复拷贝 size 个字节。repeat 可以大于等于 0。size 也可以大于等于 0，但不能超过 8，如果超过 8，也只取 8。把 repeat 个字节以 8 个为一组，每组的最高 4 个字节内容为 0，最低 4 字节内容置为 value。

size 和 value 为可选项。如果第 2 个逗号和 value 值不存在，则假定 value 为 0。如果第 1 个逗号和 size 不存在，则假定 size 为 1。

例如，在 Linux 初始化的过程中，对全局描述符表 GDT 进行设置的最后一句为：

```
.fill NR_CPUS*4,8,0    /* space for TSS's and LDT's */
```

因为每个描述符正好占 8 个字节，因此，.fill 给每个 CPU 留有存放 4 个描述符的位置。

（4）.globl symbol

.globl 使得连接程序（ld）能够看到 symbl。如果你的局部程序中定义了 symbl，那么，与这个局部程序连接的其他局部程序也能存取 symbl，例如：

```
.globl SYMBOL_NAME (idt)
.globl SYMBOL_NAME (gdt)
```

定义 idt 和 gdt 为全局符号。

（5）.quad bignums

.quad 表示零个或多个 bignums（用逗号分隔），对于每个 bignum，其缺省值是 8 字节整数。如果 bignum 超过 8 字节，则打印一个警告信息；并只取 bignum 最低 8 字节。

例如，对全局描述符表的填充就用到这个指令：

```
.quad 0x00cf9a000000ffff   /* 0x10 kernel 4GB code at 0x00000000 */
.quad 0x00cf92000000ffff   /* 0x18 kernel 4GB data at 0x00000000 */
.quad 0x00cffa000000ffff   /* 0x23 user   4GB code at 0x00000000 */
.quad 0x00cff2000000ffff   /* 0x2b user   4GB data at 0x00000000 */
```

（6）.rept count

把.rept 指令与.endr 指令之间的行重复 count 次，例如

```
    .rept   3
    .long   0
```

```
    .endr
```

相当于

```
    .long   0
    .long   0
    .long   0
```

(7) .space size , fill

这个指令保留 size 个字节的空间，每个字节的值为 fill。size 和 fill 都是常量表达式。如果逗号和 fill 被省略，则假定 fill 为 0，例如在 arch/i386/bootl/setup.S 中有一句：

```
.space  1024
```

表示保留 1024 字节的空间，并且每个字节的值为 0。

(8) .word expressions

这个表达式表示任意一节中的一个或多个表达式（用逗号分开），表达式的值占两个字节，例如：

```
gdt_descr:
.word GDT_ENTRIES*8-1
```

表示变量 gdt_descr 的值为 GDT_ENTRIES*8-1

(9) .long expressions

这与.word 类似

(10) .org new-lc , fill

把当前节的位置计数器提前到 new-lc（New Location Counter)。new-lc 或者是一个常量表达式，或者是一个与当前子节处于同一节的表达式。也就是说，你不能用.org 横跨节：如果 new-lc 是个错误的值，则.org 被忽略。.org 只能增加位置计数器的值，或者让其保持不变；但绝不能用.org 来让位置计数器倒退。

注意，位置计数器的起始值是相对于一个节的开始的，而不是子节的开始。当位置计数器被提升后，中间位置的字节被填充值 fill（这也是一个常量表达式)。如果逗号和 fill 都省略，则 fill 的缺省值为 0。

例如：.org 0x2000

ENTRY（pg0)

表示把位置计数器置为 0x2000，这个位置存放的就是临时页表 pg0。

2.6.3　gcc 嵌入式汇编

在 Linux 的源代码中，有很多 C 语言的函数中嵌入一段汇编语言程序段，这就是 gcc 提供的“ asm ”功能，例如在 include/asm-i386/system.h 中定义的，读控制寄存器 CR0 的一个宏 read_cr0 ()：

```
#define read_cr0 ()   ({ \
    unsigned int __dummy; \
    __asm__ ( \
  "movl %%cr0,%0\n\t" \
  :"=r"  (__dummy) ) ; \
    __dummy; \
```

})

这种形式看起来比较陌生，这是因为这不是标准 C 所定义的形式，而是 gcc 对 C 语言的扩充。其中__dummy 为 C 函数所定义的变量；关键词__asm__表示汇编代码的开始。括弧中第一个引号中为汇编指令 movl，紧接着有一个冒号，这种形式阅读起来比较复杂。

一般而言，嵌入式汇编语言片段比单纯的汇编语言代码要复杂得多，因为这里存在怎样分配和使用寄存器，以及把 C 代码中的变量应该存放在哪个寄存器中。为了达到这个目的，就必须对一般的 C 语言进行扩充，增加对编译器的指导作用，因此，嵌入式汇编看起来晦涩而难以读懂。

1．嵌入式汇编的一般形式

```
__asm__ __volatile__ ("<asm routine>" : output : input : modify) ;
```

其中，__asm__表示汇编代码的开始，其后可以跟__volatile__（这是可选项），其含义是避免“ asm ”指令被删除、移动或组合；然后就是小括弧，括弧中的内容是我们介绍的重点。

- "<asm routine>"为汇编指令部分，例如，"movl %%cr0,%0\n\t"。数字前加前缀“ % “，如%1，%2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数量，如 Intel 可以有 8 个。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的处理。由于这些样板操作数的前缀使用了“ % ”，因此，在用到具体的寄存器时就在前面加两个“ % ”，如%%cr0。
- 输出部分（output)，用以规定对输出变量（目标操作数）如何与寄存器结合的约束（constraint）,输出部分可以有多个约束，互相以逗号分开。每个约束以“ ＝ ”开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：

```
:"=r" (__dummy)
```

“ ＝r ”表示相应的目标操作数（指令部分的%0）可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

```
:“ ＝m ” (__dummy)
```

“ ＝m ”就表示相应的目标操作数是存放在内存单元__dummy 中。

表示约束条件的字母很多，表 2.5 给出了几个主要的约束字母及其含义。

表 2.5　　主要的约束字母及其含义

字母	含义
m, v,o	表示内存单元
R	表示任何通用寄存器
Q	表示寄存器 eax、ebx、ecx、edx 之一
I, h	表示直接操作数
E, F	表示浮点数
G	表示“ 任意 ”
a, b.c d	表示要求使用寄存器 eax/ax/al, ebx/bx/bl, ecx/cx/cl 或 edx/dx/dl

S，D	表示要求使用寄存器 esi 或 edi
I	表示常数（0～31）

输入部分（Input）：输入部分与输出部分相似，但没有“＝”。如果输入部分一个操作数所要求使用的寄存器，与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号（如“1”,“2”等）放在约束条件中，在后面的例子中，我们会看到这种情况。

修改部分（modify）:这部分常常以“memory”为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。

注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，分号“：”要保留，当“memory”存在时，三个分号都要保留，例如 system.h 中的宏定义__cli（）：

```
#define __cli ()   __asm__ __volatile__ ("cli": : :"memory")
```

2．Linux 源代码中嵌入式汇编举例

Linux 源代码中，在 arch 目录下的.h 和.c 文件中，很多文件都涉及嵌入式汇编，下面以 system.h 中的 C 函数为例，说明嵌入式汇编的应用。

（1）简单应用

```
#define __save_flags (x)     __asm__ __volatile__ ("pushfl ; popl %0":"=g" (x) : /* no input */)
#define __restore_flags (x)  __asm__ __volatile__ ("pushl %0 ; popfl": /* no output */
 :"g" (x) :"memory", "cc")
```

第 1 个宏是保存标志寄存器的值,第 2 个宏是恢复标志寄存器的值。第 1 个宏中的 pushfl 指令是把标志寄存器的值压栈。而 popl 是把栈顶的值（刚压入栈的 flags）弹出到 x 变量中，这个变量可以存放在一个寄存器或内存中。这样，你可以很容易地读懂第 2 个宏。

（2）较复杂应用

```
static inline unsigned long get_limit (unsigned long segment)
{
    unsigned long __limit;
    __asm__ ("lsll %1,%0"
  :"=r" (__limit) :"r" (segment) ) ;
   return __limit+1;
}
```

这是一个设置段界限的函数，汇编代码段中的输出参数为__limit（即%0），输入参数为 segment（即%1）。lsll 是加载段界限的指令，即把 segment 段描述符中的段界限字段装入某个寄存器（这个寄存器与__limit 结合），函数返回__limit 加 1，即段长。

（3）复杂应用

在 Linux 内核代码中，有关字符串操作的函数都是通过嵌入式汇编完成的，因为内核及用户程序对字符串函数的调用非常频繁，因此，用汇编代码实现主要是为了提高效率（当然是以牺牲可读性和可维护性为代价的）。在此，我们仅列举一个字符串比较函数 strcmp，其代码在 arch/i386／string.h 中。

```
static inline int strcmp (const char * cs,const char * ct)
{
int d0, d1;
register int __res;
__asm__ __volatile__ (
    "1:\tlodsb\n\t"
    "scasb\n\t"
    "jne 2f\n\t"
    "testb %%al,%%al\n\t"
    "jne 1b\n\t"
    "xorl %%eax,%%eax\n\t"
    "jmp 3f\n"
    "2:\tsbbl %%eax,%%eax\n\t"
    "orb $1,%%al\n"
    "3:"
    :"=a" (__res), "=&S" (d0), "=&D" (d1)
  :"1" (cs),"2" (ct));
return __res;
}
```

其中的“\n”是换行符，“\t”是 tab 符，在每条命令的结束加这两个符号，是为了让 gcc 把嵌入式汇编代码翻译成一般的汇编代码时能够保证换行和留有一定的空格。例如，上面的嵌入式汇编会被翻译成：

```
1：   lodsb   //装入串操作数，即从[esi]传送到 al 寄存器，然后 esi 指向串中下一个元素
 scasb//扫描串操作数，即从 al 中减去 es:[edi]，不保留结果，只改变标志
 jne2f//如果两个字符不相等，则转到标号 2
 testb %al  %al
 jne 1b
 xorl %eax %eax
 jmp 3f
2:   sbbl %eax %eax
 orb $1 %al
3:
```

这段代码看起来非常熟悉，读起来也不困难。其中 3f 表示往前（forword）找到第一个标号为 3 的那一行，相应地，1b 表示往后找。其中嵌入式汇编代码中输出和输入部分的结合情况为：

- 返回值__res，放在 al 寄存器中，与%0 相结合；
- 局部变量 d0，与%1 相结合，也与输入部分的 cs 参数相对应，也存放在寄存器 ESI 中，即 ESI 中存放源字符串的起始地址。
- 局部变量 d1，与%2 相结合，也与输入部分的 ct 参数相对应，也存放在寄存器 EDI 中，即 EDI 中存放目的字符串的起始地址。

通过对这段代码的分析我们应当体会到，万变不利其本，嵌入式汇编与一般汇编的区别仅仅是形式，本质依然不变。因此，全面掌握 Intel 386 汇编指令乃突破阅读低层代码之根本。

2.6.4 Intel386 汇编指令摘要

在阅读 Linux 源代码时，你可能遇到很多汇编指令，有些是你熟悉的，有些可能不熟悉，在此简要列出一些常用的 386 汇编指令及其功能。

1．位操作指令

指令	功能
BT	位测试
BTC	位测试并求反
BTR	位测试并复位
BTS	位测试并置位

2．控制转移类指令

指令	功能
CALL	调用过程
JMP	跳转
LOOP	用 ECX 计数器的循环
LOOPNZ/LOOPNE	用 ECX 计数器且不为 0 的循环／用 ECX 计数器且不等的循环
RET	返回

3．数据传输指令

指令	功能
IN	从端口输入
LEA	装入有效地址
MOV	传送
OUT	从段口输出
POP	从堆栈中弹出
POPA／POPAD	从栈弹出至所有寄存器
PUSH	压栈
PUSH／PUSHAD	所有通用寄存器压栈
XCHG	交换

4．标志控制类指令

指令	功能
CLC	清 0 进位标志
CLD	清 0 方向标志
CLI	清 0 中断标志
LAHF	将标志寄存器装入 AH 寄存器
POPF／POPFD	从栈中弹出至标志位

PUSHF／PUSHFD	将标志压栈
SAHF	将 AH 寄存器存入标志寄存器
STC	置进位标志
STD	置方向标志
STI	置中断标志

5．逻辑类指令

指令	功能
NOT	与
AND	非
OR	或
SAL/SHL	算术左移／逻辑左移
SAR	算术右移
SHLD	逻辑右移
TEST	逻辑比较
XOR	异或

6．串操作指令

指令	功能
CMPS/CMPSB/CMPSW/CMPSD	比较串操作数
INS/INSB/INSW/INSD	输入串操作数
LODS/LODSB/LODSW/LODSD	装入串操作数
MOVS/MOVSB/MOVSW/MOVSD	传送串操作数
REP	重复
REPE/REPZ	相等时重复／为 0 时重复
SCAS/SCASB/SCASW/SCASD	扫描串操作数
STOS/STOSB/STOSW/STOSD	存储串操作数

7．多段类操作指令

指令	功能
CALL	过程调用
INT	中断过程的调用
INTO	溢出中断过程的调用
IRET	中断返回
JMP	跳转
LDS	将指针转入 DS
LES	将指针转入 ES
LFS	将指针转入 FS
LGS	将指针转入 GS

LSS	将指针转入 SS
MOV	段寄存器的传送
POP	从栈弹出至段寄存器
PUSH	压栈
RET	返回

8．操作系统类指令

指令	功能
APPL	调整请求特权级
ALTS	清任务切换标志
HLT	暂停
LAR	加载访问权
LGDT	加载全局描述符表
LIDT	加载中断描述符表
LLDT	加载局部描述符表
LMSW	加载机器状态字
LSL	加载段界限
LTR	加载任务寄存器
MOV	特殊寄存器的数据传送
SGDT	存储全局描述符表
SIDT	存储中断描述符表
SMSW	存储机器状态字
STR	存储任务寄存器

第三章　中断机制

中断控制是计算机发展中一种重要的技术。最初它是为克服对 I/O 接口控制采用程序查询所带来的处理器低效率而产生的。中断控制的主要优点是只有在 I/O 需要服务时才能得到处理器的响应，而不需要处理器不断地进行查询。由此，最初的中断全部是对外部设备而言的，即称为外部中断（或硬件中断）。

但随着计算机系统结构的不断改进以及应用技术的日益提高，中断的适用范围也随之扩大，出现了所谓的内部中断（或叫异常），它是为解决机器运行时所出现的某些随机事件及编程方便而出现的。因而形成了一个完整的中断系统。

本章主要讨论在 Intel i386 保护模式下中断机制在 Linux 中的实现。如果你曾有过在 DOS 实模式下编写中断程序的经历，那么，你会发现中断在 Linux 中的实现较为复杂，这是由保护模式的复杂性引起的，因此，必须首先了解硬件机制对中断的支持。不过，不管在实模式下还是保护模式下，有关中断实现的基本原理是完全相同的。

3.1　中断基本知识

大多数读者可能对 16 位实地址模式下的中断机制有所了解，例如中断向量、外部 I/O 中断以及异常，这些内容在 32 位的保护模式下依然有效。两种模式之间最本质的差别就是在保护模式引入的中断描述符表。

3.1.1　中断向量

Intel x86 系列微机共支持 256 种向量中断，为使处理器较容易地识别每种中断源，将它们从 0～256 编号，即赋予一个中断类型码 *n*，Intel 把这个 8 位的无符号整数叫做一个向量，因此，也叫中断向量。所有 256 种中断可分为两大类：异常和中断。异常又分为故障（Fault）和陷阱（Trap），它们的共同特点是既不使用中断控制器，又不能被屏蔽。中断又分为外部可屏蔽中断（INTR）和外部非屏蔽中断（NMI），所有 I/O 设备产生的中断请求（IRQ）均引起屏蔽中断，而紧急的事件（如硬件故障）引起的故障产生非屏蔽中断。

非屏蔽中断的向量和异常的向量是固定的，而屏蔽中断的向量可以通过对中断控制器的编程来改变。Linux 对 256 个向量的分配如下。

- 从 0~31 的向量对应于异常和非屏蔽中断。
- 从 32~47 的向量（即由 I/O 设备引起的中断）分配给屏蔽中断。
- 剩余的从 48~255 的向量用来标识软中断。Linux 只用了其中的一个（即 128 或 0x80

向量）用来实现系统调用。当用户态下的进程执行一条 int 0x80 汇编指令时，CPU 就切换到内核态，并开始执行 system_call（ ）内核函数。

3.1.2 外设可屏蔽中断

Intel x86 通过两片中断控制器 8259A 来响应 15 个外中断源，每个 8259A 可管理 8 个中断源。第 1 级（称主片）的第 2 个中断请求输入端，与第 2 级 8259A（称从片）的中断输出端 INT 相连，如图 3.1 所示。我们把与中断控制器相连的每条线叫做中断线，要使用中断线，就得进行中断线的申请，就是 IRQ（Interrupt ReQuirement ），我们也常把申请一条中断线称为申请一个 IRQ 或者是申请一个中断号。IRQ 线是从 0 开始顺序编号的，因此，第一条 IRQ 线通常表示成 IRQ0。IRQn 的缺省向量是 *n*+32；如前所述，IRQ 和向量之间的映射可以通过中断控制器端口来修改。

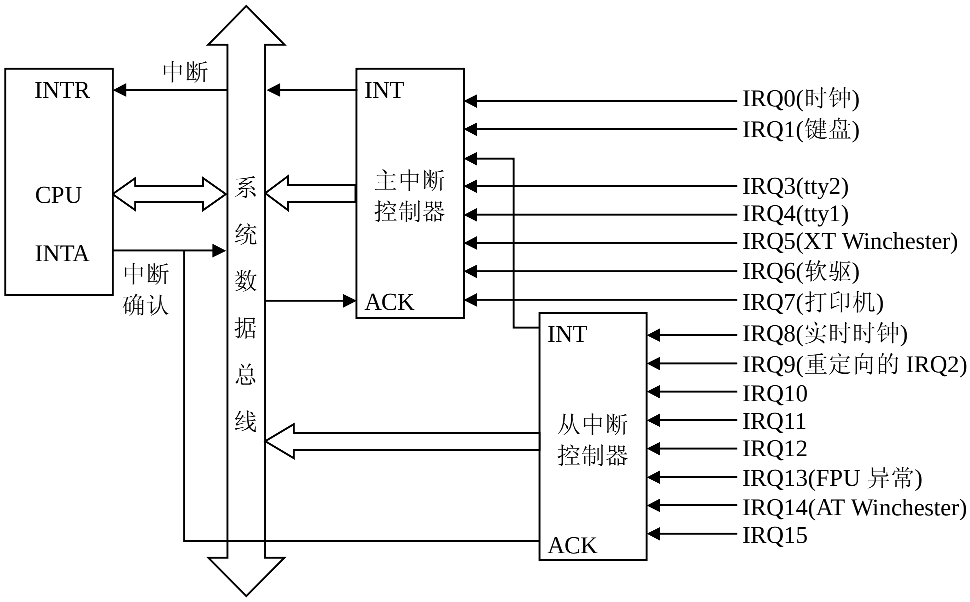


图 3.1 级连的 8259A 的中断机构

并不是每个设备都可以向中断线上发中断信号，只有对某一条确定的中断线拥有了控制权，才可以向这条中断线上发送信号。由于计算机的外部设备越来越多，所以 15 条中断线已经不够用了。中断线是非常宝贵的资源，只有当设备需要中断的时候才申请占用一个 IRQ，或者是在申请 IRQ 时采用共享中断的方式，这样可以让更多的设备使用中断。

中断控制器 8259A 执行如下操作。

（1）监视中断线，检查产生的中断请求（IRQ）信号。

（2）如果在中断线上产生了一个中断请求信号。

a. 把接受到的 IRQ 信号转换成一个对应的向量。

b. 把这个向量存放在中断控制器的一个 I/O 端口，从而允许 CPU 通过数据总线读此向量。

c. 把产生的信号发送到 CPU 的 INTR 引脚——即发出一个中断。

d. 等待，直到 CPU 确认这个中断信号，然后把它写进可编程中断控制器（PIC）的一个 I/O 端口；此时，清 INTR 线。

（3）返回到第一步。

对于外部 I/O 请求的屏蔽可分为两种情况，一种是从 CPU 的角度，也就是清除 eflag 的中断标志位（IF），当 IF=0 时，禁止任何外部 I / O 的中断请求，即关中断；一种是从中断控制器的角度，因为中断控制器中有一个 8 位的中断屏蔽寄存器（IMR），每位对应 8259A 中的一条中断线，如果要禁用某条中断线，则把 IRM 相应的位置 1，要启用，则置 0。

3.1.3　异常及非屏蔽中断

异常就是 CPU 内部出现的中断，也就是说，在 CPU 执行特定指令时出现的非法情况。非屏蔽中断就是计算机内部硬件出错时引起的异常情况。从图 3.1 可以看出，二者与外部 I/O 接口没有任何关系。Intel 把非屏蔽中断作为异常的一种来处理，因此，后面所提到的异常也包括了非屏蔽中断。在 CPU 执行一个异常处理程序时，就不再为其他异常或可屏蔽中断请求服务，也就是说，当某个异常被响应后，CPU 清除 eflag 的中 IF 位，禁止任何可屏蔽中断。但如果又有异常产生，则由 CPU 锁存（CPU 具有缓冲异常的能力），待这个异常处理完后，才响应被锁存的异常。我们这里讨论的异常中断向量在 0～31 之间，不包括系统调用（中断向量为 0x80）。

Intel x86 处理器发布了大约 20 种异常（具体数字与处理器模式有关）。Linux 内核必须为每种异常提供一个专门的异常处理程序。这里特别说明的是，在某些异常处理程序开始执行之前，CPU 控制单元会产生一个硬件错误码，内核先把这个错误码压入内核栈中。

在表 3.1 中给出了 Pentium 模型中异常的向量、名字、类型及简单描述。更多的信息可以在 Intel 的技术文挡中找到。

表 3.1　　　　异常的简单描述

向量	异常名	类别	描述
0	除法出错	故障	被 0 除
1	调试	故障 / 陷阱	当对一个程序进行逐步调试时
2	非屏蔽中断（NMI）	为不可屏蔽中断保留	
3	断点	陷阱	由 int3（断点指令）指令引起
4	溢出	陷阱	当 into（check for overflow）指令被执行
5	边界检查	故障	当 bound 指令被执行
6	非法操作码	故障	当 CPU 检查到一个无效的操作码
7	设备不可用	故障	随着设置 cr0 的 TS 标志，ESCAPE 或 MMX 指令被执行
8	双重故障	故障	处理器不能串行处理异常而引起的

续表

向量	异常名	类别	描述
9	协处理器段越界	故障	因外部的数学协处理器引起的问题（仅用在 80386）
10	无效 TSS	故障	要切换到的进程具有无效的 TSS
11	段不存在	故障	引用一个不存在的内存段
12	栈段异常	故障	试图超过栈段界限，或由 ss 标识的段不在内存
13	通用保护	故障	违反了 Intelx86 保护模式下的一个保护规则
14	页异常	故障	寻址的页不在内存，或违反了一种分页保护机制
15	Intel 保留	/	保留
16	浮点出错	故障	浮点单元用信号通知一个错误情形，如溢出
17	对齐检查	故障	操作数的地址没有被正确地排列

18～31 由 Intel 保留，为将来的扩充用。

另外，如表 3.2 所示，每个异常都由专门的异常处理程序来处理（参见本章后面“异常处理“部分），它们通常把一个 UNIX 信号发送到引起异常的进程。

表 3.2　　由异常处理程序发送的信号

向量	异常名	出错码	异常处理程序	信号
0	除法出错	/	divide_error（ ）	SIGFPE
1	调试	/	debug（ ）	SIGTRAP
2	非屏蔽中断（NMI）	/	nmi（ ）	None
3	断点	/	int3（ ）	SIGTRAP
4	溢出	/	overflow（ ）	SIGSEGV
5	边界检查	/	bounds（ ）	SIGSEGV
6	非法操作码	/	invalid_op（ ）	SIGILL
7	设备不可用	/	device_not_available（ ）	SIGSEGV
8	双重故障	有	double_fault（ ）	SIGSEGV
9	协处理器段越界	/	coprocessor_segment_overrun（ ）	SIGFPE
10	无效 TSS	有	invalid_tss（ ）	SIGSEGV
11	段不存在	有	segment_not_present（ ）	SIGBUS
12	栈段异常	有	stack_segment（ ）	SIGBUS
13	通用保护	有	general_protection（ ）	SIGSEGV
14	页异常	有	page_fault（ ）	SIGSEGV

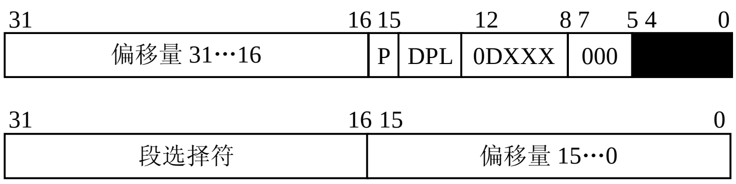
15	Intel 保留	/	None	None

续表

向量	异常名	出错码	异常处理程序	信号
16	浮点出错	/	coprocessor_error（　）	SIGFPE
17	对齐检查	/	alignment_check（　）	SIGSEGV

3.1.4　中断描述符表

在实地址模式中，CPU 把内存中从 0 开始的 1K 字节作为一个中断向量表。表中的每个表项占 4 个字节，由两个字节的段地址和两个字节的偏移量组成，这样构成的地址便是相应中断处理程序的入口地址。但是，在实模式下，由 4 字节的表项构成的中断向量表显然满足不了要求。这是因为，①除了两个字节的段描述符，偏移量必用 4 字节来表示；②要有反映模式切换的信息。因此，在实模式下，中断向量表中的表项由 8 个字节组成，如图 3.2 所示，中断向量表也改叫做中断描述符表 IDT（Interrupt Descriptor Table）。其中的每个表项叫做一个门描述符（Gate Descriptor），“门”的含义是当中断发生时必须先通过这些门，然后才能进入相应的处理程序。



DPL	段描述符的特权级
偏移量	入口函数地址的偏移量
P	段是否在内存中的标志
段选择符	入口函数所处代码段的选择符
D	标志位，1=32 位，0=16 位
XXX	3 位门类型码

图 3.2　门描述符的一般格式

其中类型占 3 位，表示门描述符的类型，这些描述符如下。

1．任务门（Task gate）

其类型码为 101，门中包含了一个进程的 TSS 段选择符，但偏移量部分没有使用，因为 TSS 本身是作为一个段来对待的，因此，任务门不包含某一个入口函数的地址。TSS 是 Intel 所提供的任务切换机制，但是 Linux 并没有采用任务门来进行任务切换（参见第五章的任务切换）。

2．中断门（Interrupt gate）

其类型码为 110，中断门包含了一个中断或异常处理程序所在段的选择符和段内偏移量。当控制权通过中断门进入中断处理程序时，处理器清 IF 标志，即关中断，以避免嵌套中断的

发生。中断门中的 DPL（Descriptor Privilege Level）为 0，因此，用户态的进程不能访问 Intel 的中断门。所有的中断处理程序都由中断门激活，并全部限制在内核态。

3．陷阱门（Trap gate）

其类型码为 111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时维持 IF 标志位不变，也就是说，不关中断。

4．系统门（System gate）

这是 Linux 内核特别设置的，用来让用户态的进程访问 Intel 的陷阱门，因此，门描述符的 DPL 为 3。通过系统门来激活 4 个 Linux 异常处理程序，它们的向量是 3、4、5 及 128，也就是说，在用户态下，可以使用 int3、into、bound 及 int0x80 四条汇编指令。

最后，在保护模式下，中断描述符表在内存的位置不再限于从地址 0 开始的地方，而是可以放在内存的任何地方。为此，CPU 中增设了一个中断描述符表寄存器 IDTR，用来存放中断描述符表在内存的起始地址。中断描述符表寄存器 IDTR 是一个 48 位的寄存器，其低 16 位保存中断描述符表的大小，高 32 位保存 IDT 的基址，如图 3.3 所示。

47　　　　16	15　　　　0
32 位基址值	界限

图 3.3 中断描述符表寄存器 IDTR

3.1.5 相关汇编指令

为了有助于读者对中断实现过程的理解，下面介绍几条相关的汇编指令。

1．调用过程指令 CALL

指令格式：CALL　过程名

说明：i386 在取出 CALL 指令之后及执行 CALL 指令之前，使指令指针寄存器 EIP 指向紧接 CALL 指令的下一条指令。CALL 指令先将 EIP 值压入栈内，再进行控制转移。当遇到 RET 指令时，栈内信息可使控制权直接回到 CALL 的下一条指令

2．调用中断过程指令 INT

指令格式：INT　中断向量

说明：EFLAG、CS 及 EIP 寄存器被压入栈内。控制权被转移到由中断向量指定的中断处理程序。在中断处理程序结束时，IRET 指令又把控制权送回到刚才执行被中断的地方。

3．调用溢出处理程序的指令 INTO

指令格式：INTO

说明：在溢出标志为 1 时，INTO 调用中断向量为 4 的异常处理程序。EFLAG、CS 及 EIP 寄存器被压入栈内。控制权被转移到由中断向量 4 指定的异常处理程序。在中断处理程序结束时，IRET 指令又把控制权送回到刚才执行被中断的地方。

4．中断返回指令 IRET

指令格式：IRET

说明：IRET 与中断调用过程相反：它将 EIP、CS 及 EFLAGS 寄存器内容从栈中弹出，并将控制权返回到发生中断的地方。IRET 用在中断处理程序的结束处。

5．加载中断描述符表的指令 LIDT

格式：LIDT　48 位的伪描述符

说明：LIDT 将指令中给定的 48 位伪描述符装入中断描述符寄存器 IDTR。伪描述符和中断描述符表寄存器的结构相同，都是由两部分组成：在低字（低 16 位）中装的是界限，在高双字（高 32 位）中装的是基址。这条指令只能出现在操作系统的代码中。

中断或异常处理程序执行的最后一条指令是返回指令 IRET。这条指令将使 CPU 进行如下操作后，把控制权转交给被中断的进程。

- 从中断处理程序的内核栈中恢复相应寄存器的值。如果一个硬件错码被压入堆栈，则先弹出这个值，然后，依次将 EIP、CS 及 EFLSG 从栈中弹出。
- 检查中断或异常处理程序的 CPL 是否等于 CS 中的最低两位，如果是，这就意味着被中断的进程与中断处理程序都处于内核态，也就是没有更换堆栈，因此，IRET 终止执行，返回到被中断的进程。否则，转入下一步。
- 从栈中装载 SS 和 ESP 寄存器，返回到用户态堆栈。
- 检查 DS、ES、FS 和 GS 四个段寄存器的内容，看它们包含的选择符是否是一个段选择符，并且其 DPL 是否小于 CPL。如果是，就清除其内容。这么做的原因是为了禁止用户态的程序（CPL=3）利用内核曾用过的段寄存器（DPL=0）。如果不这么做，怀有恶意的用户就可能利用这些寄存器来访问内核的地址空间。

3.2　中断描述符表的初始化

通过上面的介绍，我们知道了 Intel 微处理器对中断和异常所做的工作。下面，我们从操作系统的角度来对中断描述符表的初始化给予描述。

Linux 内核在系统的初始化阶段要进行大量的初始化工作，其与中断相关的工作有：初始化可编程控制器 8259A；将中断向量 IDT 表的起始地址装入 IDTR 寄存器，并初始化表中的每一项。这些操作的完成将在本节进行具体描述。

用户进程可以通过 INT 指令发出一个中断请求，其中断请求向量在 0～255 之间。为了防止用户使用 INT 指令模拟非法的中断和异常，必须对 IDT 表进行谨慎的初始化。其措施之一就是将中断门或陷阱门中的 DPL 域置为 0。如果用户进程确实发出了这样一个中断请求，CPU 会检查出其 CPL（3）与 DPL（0）有冲突，因此产生一个“通用保护”异常。

但是，有时候必须让用户进程能够使用内核所提供的功能（比如系统调用），也就是说从用户空间进入内核空间，这可以通过把中断门或陷阱门的 DPL 域置为 3 来达到。

3.2.1　外部中断向量的设置

前面我们已经提到，Linux 把向量 0～31 分配给异常和非屏蔽中断，而把 32～47 之间的向量分配给可屏蔽中断，可屏蔽中断的向量是通过对中断控制器的编程来设置的。前面介绍了 8259A 中断控制器，下面我们通过对其初始化的介绍，来了解如何设置中断向量。

8259A 通过两个端口来进行数据传送，对于单块的 8259A 或者是级连中的 8259A_1 来说，这两个端口是 0x20 和 0x21。对于 8259A_2 来说，这两个端口是 0xA0 和 0xA1。8259A 有两种编程方式，一是初始化方式，二是工作方式。在操作系统启动时，需要对 8959A 做一些初始化工作，这就是初始化方式编程。

先简单介绍一下 8259A 内部的 4 个中断命令字（ICW）寄存器的功能，它们都是用来启动初始化编程的。

- ICW1：初始化命令字。
- ICW2：中断向量寄存器，初始化时写入高 5 位作为中断向量的高五位，然后在中断响应时由 8259 根据中断源（哪个管脚）自动填入形成完整的 8 位中断向量（或叫中断类型号）。
- ICW3：8259 的级连命令字，用来区分主片和从片。
- ICW4：指定中断嵌套方式、数据缓冲选择、中断结束方式和 CPU 类型。

8259A 初始化的目的是写入有关命令字，8259A 内部有相应的寄存器来锁存这些命令字，以控制 8259A 工作。有关的硬件知识笔者就不详细描述了，请读者查阅有关可编程中断控制器的资料，我们只具体把 Linux 对 8259A 的初始化讲解一下，代码在 /arch/i386/kernel/i8259.c 的函数 init_8259A()中:

```
      outb（0xff, 0x21）；        /* 送数据到工作寄存器 OCW1（又称中断屏蔽字），
        屏蔽所有外部中断, 因为此时系统尚未初始化完毕,
        outb（0xff, 0xA1）；        不能接收任何外部中断请求  */


            outb_p（0x11, 0x20）； /*送 0x11 到 ICW1（通过端口 0x20），启动初始化编
程。0x11 表示外部中断请求信号为上升沿有效，系统中有多片 8295A 级连，还表示要向 ICW4
送数据 */


            outb_p（0x20 + 0, 0x21）; /* 送 0x20 到 ICW2，写入高 5 位作为中断向量的
高 5 位，低 3 位根据中断源（管脚）填入中断号 0～7，因此把 IRQ0-7 映射到向量 0x20-0x27
*/


            outb_p（0x04, 0x21）;/*送 0x04 到 ICW3，ICW3 是 8259 的级连命令字， 0x04
表示 8259A-1 是主片 */
            outb_p（0x11, 0xA0）;  /* 用 ICW1 初始化 8259A-2 */
            outb_p（0x20 + 8, 0xA1）; /* 用 ICW2 把 8259A-2 的 IRQ0-7 映射到 0x28-0x2f
*/
            outb_p（0x02, 0xA1）;   /* 送 0x04 到 ICW3。表示 8259A-2 是从片，并连
```

接在 8259A_1 的 2 号管脚上*/

```
outb_p（0x01, 0xA1）; /* 把 0x01 送到 ICW4 */
```

最后一句有 4 方面含义：①中断嵌套方式为一般嵌套方式。当某个中断正在服务时，本级中断及更低级的中断都被屏蔽，只有更高级的中断才能响应。注意，这对于多片 8259A 级连的中断系统来说，当某从片中一个中断正在服务时，主片即将这个从片的所有中断屏蔽，所以此时即使本片有比正在服务的中断级别更高的中断源发出请求，也不能得到响应，即不能中断嵌套。②8259A 数据线和系统总线之间不加三态缓冲器。一般来说，只有级连片数很多时才用到三态缓冲器；③中断结束方式为正常方式（非自动结束方式）。即在中断服务结束时（中断服务程序末尾），要向 8259A 芯片发送结束命令字 EOI（送到工作寄存器 OCW2 中），于是中断服务寄存器 ISR 中的当前服务位被清 0，EOI 命令字的格式有多种，在此不详述；④CPU 类型为 x86 系列。

outb_p()函数就是把第一个操作数拷贝到由第二个操作数指定的 I/O 端口，并通过一个空操作来产生一个暂停。

这里介绍了 8259A 初始化的主要工作。最后要说明的是：IBM PC 机的 BIOS 中固化有对中断控制器的初始化程序段，在计算机加电时，这段程序自动执行，读者感兴趣可以查阅资料看看它的源代码。典型的 PC 机将外部中断的中断向量分配为：08H~0FH，70H~77H。但是 Linux 对 8259A 作了重新初始化，修改了外部中断的中断向量的分配（20H~2FH），使中断向量的分配更加合理。

3.2.2 中断描述符表 IDT 的预初始化

当计算机运行在实模式时，IDT 被初始化并由 BIOS 使用。然而，一旦真正进入了 Linux 内核，IDT 就被移到内存的另一个区域，并进行进入实模式的初步初始化。

1. 中断描述表寄存器 IDTR 的初始化

用汇编指令 LIDT 对中断向量表寄存器 IDTR 进行初始化，其代码在 arch/i386/boot/setup.S 中：

```
lidt    idt_48                  # load idt with 0,0
…
    idt_48:
        .word   0                  # idt limit = 0
        .word   0, 0                # idt base = 0L
```

2. 把 IDT 表的起始地址装入 IDTR

用汇编指令 LIDT 装入 IDT 的大小和它的地址（在 arch/i386/kernel/head.S 中）：

```
#define IDT_ENTRIES     256
.globl SYMBOL_NAME (idt)

lidt idt_descr
…
idt_descr:
       .word IDT_ENTRIES*8-1           # idt contains 256 entries
```

```
SYMBOL_NAME（idt）:
        .long SYMBOL_NAME（idt_table）
```

其中 idt 为一个全局变量，内核对这个变量的引用就可以获得 IDT 表的地址。表的长度为 256×8＝2048 字节。

3．用 setup_idt()函数填充 idt_table 表中的 256 个表项

我们首先要看一下 idt_table 的定义（在 arch/i386/kernel/traps.c 中）：

```
  struct desc_struct idt_table[256] __attribute__（（__section__（".data.idt"）））  ={{0, 0}, };
```

desc_struct 结构定义为：

```
   struct desc_struct  {
    unsigned long a,b   }
```

对 idt_table 变量还定义了其属性（__attribute__），__section__是汇编中的“节”，指定了 idt_table 的起始地址存放在数据节的 idt 变量中，如上面第“2.把 IDT 表的起始地址装入 IDTR”所述。

在对 idt_table 表进行填充时，使用了一个空的中断处理程序 ignore_int()。因为现在处于初始化阶段，还没有任何中断处理程序，因此用这个空的中断处理程序填充每个表项。ignore_int()是一段汇编程序（在 head.S 中）：

```
ignore_int:
         cld        #方向标志清 0，表示串指令自动增长它们的索引寄存器（esi 和 edi）
         pushl %eax
         pushl %ecx
         pushl %edx
         pushl %es
         pushl %ds
         movl $（__KERNEL_DS）,%eax
         movl %eax,%ds
         movl %eax,%es
         pushl $int_msg
         call SYMBOL_NAME（printk）
         popl %eax
         popl %ds
         popl %es
         popl %edx
         popl %ecx
         popl %eax
         iret
int_msg:
        .asciz "Unknown interrupt\n"
         ALIGN
```

该中断处理程序模仿一般的中断处理程序，执行如下操作：

- 在栈中保存一些寄存器的值；
- 调用 printk()函数打印“Unknown interrupt”系统信息；

- 从栈中恢复寄存器的内容；
- 执行 iret 指令以恢复被中断的程序。

实际上，ignore_int()处理程序应该从不执行。如果在控制台或日志文件中出现了“Unknown interrupt”消息，说明要么是出现了一个硬件问题（一个 I/O 设备正在产生没有预料到的中断），要么就是出现了一个内核问题（一个中断或异常未被恰当地处理）。

最后，我们来看 setup_idt()函数如何对 IDT 表进行填充:

```
/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It doesn't actually load
 *  idt - that can be done only after paging has been enabled
 *  and the kernel moved to PAGE_OFFSET. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok.
 */
setup_idt:
        lea ignore_int,%edx    /*计算 ignore_int 地址的偏移量,并将其装入%edx*/
        movl $（__KERNEL_CS << 16）,%eax   /* selector = 0x0010 = cs */
        movw %dx,%ax
        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */

        lea SYMBOL_NAME（idt_table）,%edi
        mov $256,%ecx
rp_sidt:
        movl %eax,（%edi）
        movl %edx,4（%edi）
        addl $8,%edi
        dec %ecx
        jne rp_sidt
        ret
```

这段程序的理解要对照门描述符的格式。8 个字节的门描述符放在两个 32 位寄存器 eax 和 edx 中，如图 3.4 所示，从 rp_sidt 开始的那段程序是循环填充 256 个表项。

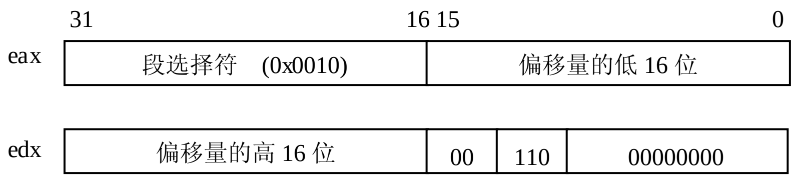


图 3.4　门描述符存放在两个 32 位的寄存器中

3.2.3　中断向量表的最终初始化

在对中断描述符表进行预初始化后，内核将在启用分页功能后对 IDT 进行第二遍初始化，也就是说，用实际的陷阱和中断处理程序替换这个空的处理程序。一旦这个过程完成，对于

每个异常，IDT 都由一个专门的陷阱门或系统门，而对每个外部中断，IDT 都包含专门的中断门。

1．IDT 表项的设置

IDT 表项的设置是通过_set_gaet()函数实现的，这与 IDT 表的预初始化比较相似，但这里使用的是嵌入式汇编，因此，理解起来比较困难。在此，我们给出函数源码（在 traps.c 中）及其解释：

```
#define _set_gate（gate_addr,type,dpl,addr） \
do { \
   int __d0, __d1; \
   __asm__ __volatile__ （"movw %%dx,%%ax\n\t" \
         "movw %4,%%dx\n\t" \
         "movl %%eax,%0\n\t" \
        "movl %%edx,%1" \
         :"=m" （*（（long *） （gate_addr））），\
          "=m" （*（1+（long *） （gate_addr））），"=&a" （__d0），"=&d" （__d1） \
         :"i" （（short） （0x8000+（dpl<<13）+（type<<8））），\
          "3" （（char *） （addr））,"2" （__KERNEL_CS << 16））; \
} while （0）
```

这是一个带参数的宏定义，其中，gate_addr 是门的地址，type 为门类型，dpl 为请求特权级，addr 为中断处理程序的地址。对这段嵌入式汇编代码的说明如下：

- 输出部分有 4 个变量，分别与％1、％2、％3 及％4 相结合，其中，％0 与 gate_addr 结合，1％与（gate_aggr＋1）结合，这两个变量都存放在内存；2％与局部变量__d0 结合，存放在 eax 寄存器中；3％与__d1 结合，存放在 edx 寄存器中。
- 输入部分有 3 个变量。由于输出部分已定义了 0%～3％，因此，输入部分的第一个变量为 4％，其值为“0x8000+（dpl<<13）+（type<<8”，而后面两个变量分别等价于输出部分的％3（edx）和 2％（eax），其值分别为“addr”和“__KERNEL_CS << 16”
- 有了参数的这种对照关系，再参考前面的 set_idt()函数，就不难理解那 4 条 mov 语句了。

下面我们来看如何调用_set_get()函数来给 IDT 插入门:

```
void set_intr_gate（unsigned int n, void *addr）
{
         _set_gate（idt_table+n,14,0,addr）;
}
```

在第 *n* 个表项中插入一个中断门。这个门的段选择符设置成代码段的选择符(__KERNEL_CS),DPL 域设置成 0,14 表示 D 标志位为 1 而类型码为 110,所以 set_intr_gate()设置的是中断门，偏移域设置成中断处理程序的地址 addr。

```
static void __init set_trap_gate（unsigned int n, void *addr）
{
         _set_gate（idt_table+n,15,0,addr）;
}
```

在第 *n* 个表项中插入一个陷阱门。这个门的段选择符设置成代码段的选择符，DPL 域设置成 0，15 表示 D 标志位为 1 而类型码为 111，所以 set_trap_gate()设置的是陷阱门，偏移域设置成异常处理程序的地址 addr。

```
static void __init set_system_gate（unsigned int n, void *addr）
```

```
{
        _set_gate (idt_table+n,15,3,addr) ;
}
```

在第 *n* 个表项中插入一个系统门。这个门的段选择符设置成代码段的选择符，DPL 域设置成 3，15 表示 D 标志位为 1 而类型码为 111，所以 set_system_gate()设置的也是陷阱门，但因为 DPL 为 3，因此，系统调用在用户空间可以通过“INT0X80”顺利穿过系统门，从而进入内核空间。

2．对陷阱门和系统门的初始化

trap_init()函数就是设置中断描述符表开头的 19 个陷阱门，如前所说，这些中断向量都是 CPU 保留用于异常处理的：

```
set_trap_gate (0,&divide_error) ;
set_trap_gate (1,&debug) ;
set_intr_gate (2,&nmi) ;
set_system_gate (3,&int3) ;      /* int3-5 can be called from all */
set_system_gate (4,&overflow) ;
set_system_gate (5,&bounds) ;
set_trap_gate (6,&invalid_op) ;
set_trap_gate (7,&device_not_available) ;
set_trap_gate (8,&double_fault) ;
set_trap_gate (9,&coprocessor_segment_overrun) ;
set_trap_gate (10,&invalid_TSS) ;
set_trap_gate (11,&segment_not_present) ;
set_trap_gate (12,&stack_segment) ;
set_trap_gate (13,&general_protection) ;
set_intr_gate (14,&page_fault) ;
set_trap_gate (15,&spurious_interrupt_bug) ;
set_trap_gate (16,&coprocessor_error) ;
set_trap_gate (17,&alignment_check) ;
set_trap_gate (18,&machine_check) ;
set_trap_gate (19,&simd_coprocessor_error) ;

set_system_gate (SYSCALL_VECTOR,&system_call) ;
```

在对陷阱门及系统门设置以后，我们来看一下中断门的设置。

3．中断门的设置

下面介绍的相关代码均在 arch/I386/kernel/i8259.c 文件中，其中中断门的设置是由 init_IRQ（ ）函数中的一段代码完成的：

```
for (i = 0; i< NR_IRQS; i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (vector != SYSCALL_VECTOR)
                set_intr_gate (vector, interrupt[i]) ;
```

其含义比较明显：从 FIRST_EXTERNAL_VECTOR 开始，设置 NR_IRQS 个 IDT 表项。常数 FIRST_EXTERNAL_VECTOR 定义为 0x20，而 NR_IRQS 则为 224，即中断门的个数。注意，必须跳过用于系统调用的向量 0x80，因为这在前面已经设置好了。

这里，中断处理程序的入口地址是一个数组 interrupt[]，数组中的每个元素是指向中

断处理函数的指针。我们来看一下编码的作者如何巧妙地避开了繁琐的文字录入，而采用统一的方式处理多个函数名。

```
#define IRQ (x,y)  \
        IRQ##x##y##_interrupt

#define IRQLIST_16 (x)  \
        IRQ (x,0) , IRQ (x,1) , IRQ (x,2) , IRQ (x,3) , \
        IRQ (x,4) , IRQ (x,5) , IRQ (x,6) , IRQ (x,7) , \
        IRQ (x,8) , IRQ (x,9) , IRQ (x,a) , IRQ (x,b) , \
        IRQ (x,c) , IRQ (x,d) , IRQ (x,e) , IRQ (x,f)
void  (*interrupt[NR_IRQS])  (void)  = IRQLIST_16 (0x0)
```

其中，“＃＃”的作用是把字符串连接在一起。经过 gcc 预处理，IRQLIST_16（0x0）被替换为 IRQ0x00_interrupt，IRQ0x01_interrupt，IRQ0x02_interrupt…IRQ0x0f_interrupt。

到此为止，我们已经介绍了 15 个陷阱门、4 个系统门和 16 个中断门的设置。内核代码中还有对其他中断门的设置，在此就不一一介绍。

3.3 异常处理

Linux 利用异常来达到两个截然不同的目的：

- 给进程发送一个信号以通报一个反常情况；
- 处理请求分页。

对于第一种情况，例如，如果进程执行了一个被 0 除的操作，CPU 则会产生一个“除法错误”异常，并由相应的异常处理程序向当前进程发送一个 SIGFPE 信号。当前进程接收到这个信号后，就要采取若干必要的步骤，或者从错误中恢复，或者终止执行（如果这个信号没有相应的信号处理程序）。

内核对异常处理程序的调用有一个标准的结构，它由以下 3 部分组成：

- 在内核栈中保存大多数寄存器的内容（由汇编语言实现）；
- 调用 C 编写的异常处理函数；
- 通过 ret_from_exception()函数从异常退出。

3.3.1 在内核栈中保存寄存器的值

所有异常处理程序被调用的方式比较相似，因此，我们用 handler_name 来表示一个通用的异常处理程序的名字（实际名字都出现在表 3.1 中）。进入异常处理程序的汇编指令在 arch/I386/kernel/entry.S 中：

```
handler_name:
    pushl $0  /* only for some exceptions */
    pushl $do_handler_name
jmp error_code

例如： overflow:
            pushl $0
```

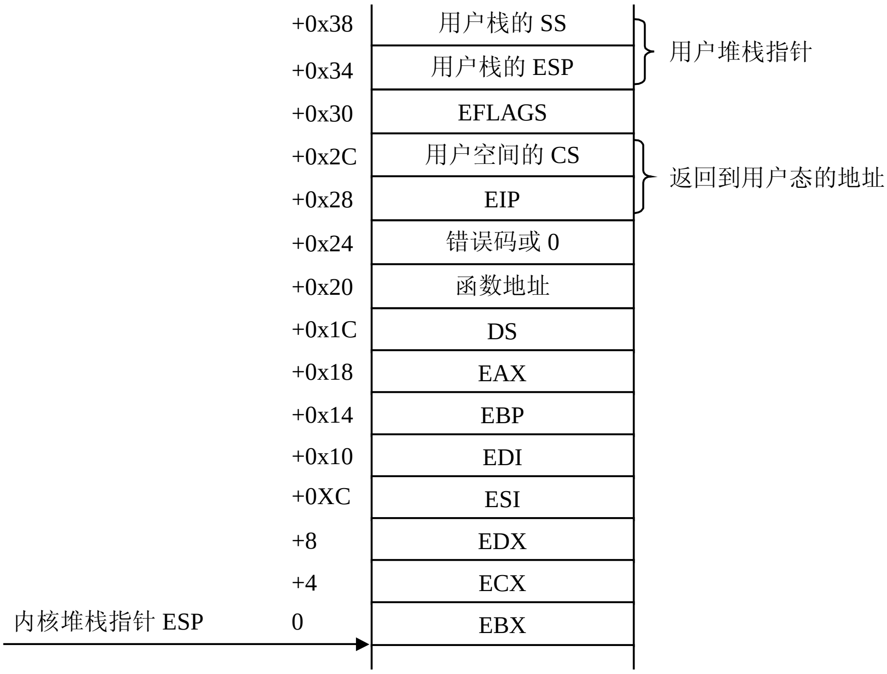
```
pushl $ do_overflow
jmp error_code
```

当异常发生时，如果控制单元没有自动地把一个硬件错误代码插入到栈中，相应的汇编语言片段会包含一条 pushl $0 指令，在栈中垫上一个空值；如果错误码已经被压入堆栈，则没有这条指令。然后，把异常处理函数的地址压进栈中，函数的名字由异常处理程序名与 do_ 前缀组成。

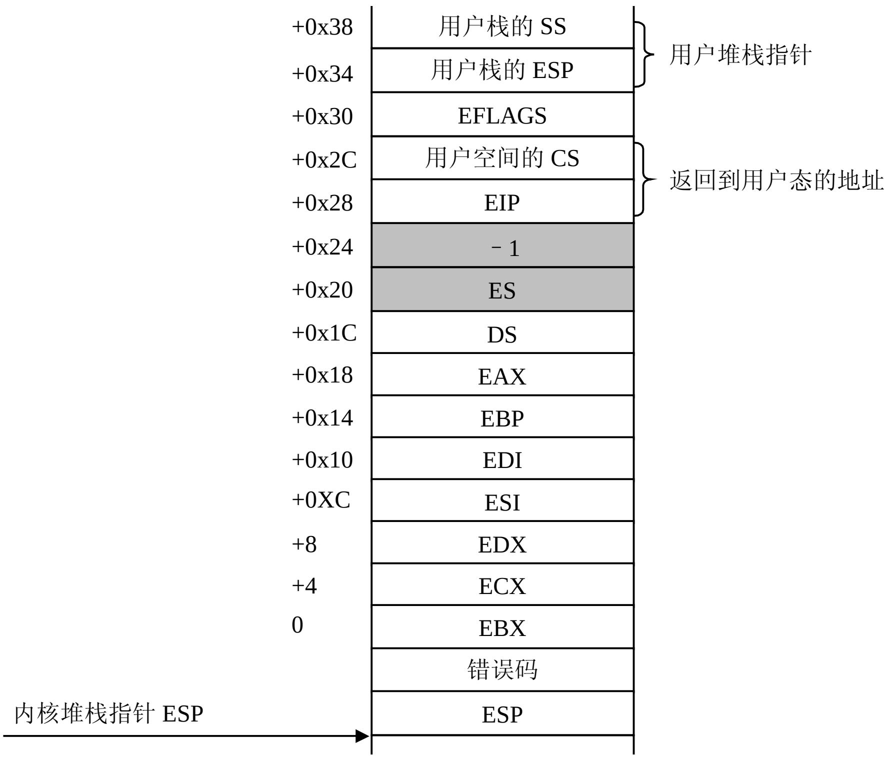
标号为 error_code 的汇编语言片段对所有的异常处理程序都是相同的，除了“设备不可用”这一个异常。这段代码实际上是为异常处理程序的调用和返回进行相关的操作，代码如下：

```
error_code:
      pushl %ds
      pushl %eax
      xorl %eax,%eax
      pushl %ebp
      pushl %edi              #把 C 函数可能用到的寄存器都保存在栈中
      pushl %esi
      pushl %edx
      decl %eax             #eax = -1
      pushl %ecx
      pushl %ebx
      cld               #  清 eflags 的方向标志，以确保 edi 和 esi 寄存器的值自动增加
      movl %es,%ecx
      movl ORIG_EAX（%esp）, %esi       # get the error code, ORIG_EAX= 0x24
      movl ES（%esp）, %edi              # get the function address, ES = 0x20
      movl %eax, ORIG_EAX（%esp）       #把栈中的这个位置置为-1
      movl %ecx, ES（%esp）
      movl %esp,%edx
      pushl %esi                     # push the error code
      pushl %edx                    # push the pt_regs pointer
      movl $（__KERNEL_DS）,%edx
      movl %edx,%ds            #把内核数据段选择符装入 ds 寄存器
      movl %edx,%es
      GET_CURRENT（%ebx）      #ebx 中存放当前进程 task_struct 结构的地址
      call *%edi                    #调用这个异常处理程序
      addl $8,%esp
     jmp ret_from_exception
```

图 3.5 给出了从用户进程进入异常处理程序时内核堆栈的变化示意图。



(a) 进入异常处理程序时内核堆栈示意图



(b) 异常处理程序被调用后堆栈的示意图

图 3.5　进入异常后内核堆栈的变化

3.3.2　中断请求队列的初始化

由于硬件的限制，很多外部设备不得不共享中断线，例如，一些 PC 配置可以把同一条中断线分配给网卡和图形卡。由此看来，让每个中断源都必须占用一条中断线是不现实的。所以，仅用中断描述符表并不能提供中断产生的所有信息，内核必须对中断线给出进一步的描述。在 Linux 设计中，专门为每个中断请求 IRQ 设置了一个队列，这就是我们所说的中断请求队列。

注意，中断线、中断请求（IRQ）号及中断向量之间的关系为：中断线是中断请求的一种物理描述，逻辑上对应一个中断请求号（或简称中断号），第 n 个中断号（IRQn）的缺省中断向量是 n+32。

3.3.3　中断请求队列的数据结构

如前所述，在 256 个中断向量中，除了 32 个分配给异常外，还有 224 个作为中断向量。对于每个 IRQ，Linux 都用一个 irq_desc_t 数据结构来描述，我们把它叫做 IRQ 描述符，224 个 IRQ 形成一个数组 irq_desc[]，其定义在/include/linux/irq.h 中：

```
 /*
  * This is the "IRQ descriptor", which contains various information
  * about the irq, including what kind of hardware handling it has,
  * whether it is disabled etc etc.
  *
  * Pad this out to 32 bytes for cache and indexing reasons.
  */
typedef struct {
         unsigned int status;            /* IRQ status */
         hw_irq_controller *handler;
         struct irqaction *action;        /* IRQ action list */
         unsigned int depth;            /* nested irq disables */
         spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];
```

编码作者对这个数据结构给出了一定的解释，“____cacheline_aligned”表示这个数据结构的存放按 32 字节（高速缓存行的大小）进行对齐，以便于将来存放在高速缓存并容易存取。下面对这个数据结构的各个域给予描述。

status

描述 IRQ 中断线状态的一组标志（在 irq.h 中定义）,其具体含义及应用将在 do_IRQ() 函数中介绍。

handler

指向 hw_interrupt_type 描述符，这个描述符是对中断控制器的描述，下面会给出具体解释。

action

指向一个单向链表的指针，这个链表就是对中断服务例程进行描述的 irqaction 结构，后面将给予具体描述。

depth

如果启用这条 IRQ 中断线，depth 则为 0，如果禁用这条 IRQ 中断线不止一次，则为一个正数。每当调用一次 disable_irq（ ），该函数就对这个域的值加 1；如果 depth 等于 0，该函数就禁用这条 IRQ 中断线。相反，每当调用 enable_irq（ ）函数时，该函数就对这个域的值减 1；如果 depth 变为 0，该函数就启用这条 IRQ 中断线。

1．IRQ 描述符的初始化

在系统初始化期间，init_ISA_irqs()函数对 IRQ 数据结构（或叫描述符）的域进行初始化（参见 i8258.c）：

```
for (i = 0; i < NR_IRQS; i++) {
            irq_desc[i].status = IRQ_DISABLED;
            irq_desc[i].action = 0;
           irq_desc[i].depth = 1;

           if (i < 16) {
                   /*
                    * 16 old-style INTA-cycle interrupts:
                    */
                    irq_desc[i].handler = &i8259A_irq_type;
           } else {
                   /*
                    * 'high' PCI IRQs filled in on demand
                   */
                    irq_desc[i].handler = &no_irq_type;
          }
    }
```

从这段程序可以看出，初始化时，让所有的中断线都处于禁用状态；每条中断线上还没有任何中断服务例程（action 为 0）；因为中断线被禁用，因此 depth 为 1；对中断控制器的描述分为两种情况，一种就是通常所说的 8259A，另一种是其他控制器。

然后，更新中断描述符表 IDT，如 3.2.3 节所述，用最终的中断门来代替临时使用的中断门。

2．中断控制器描述符 hw_interrupt_type

这个描述符包含一组指针，指向与特定中断控制器电路（PIC）打交道的低级 I/O 例程，定义如下：

```
/*
* Interrupt controller descriptor. This is all we need
* to describe about the low-level hardware.
*/
struct hw_interrupt_type {
        const char * typename;
        unsigned int (*startup) (unsigned int irq);
        void (*shutdown) (unsigned int irq);
       void (*enable) (unsigned int irq);
       void (*disable) (unsigned int irq);
       void (*ack) (unsigned int irq);
       void (*end) (unsigned int irq);
```

```
        void (*set_affinity) (unsigned int irq, unsigned long mask);
};

typedef struct hw_interrupt_type  hw_irq_controller;
```

Linux 除了支持本章前面已提到的 8259A 芯片外，也支持其他的 PIC 电路，如 SMP IO-APIC、PIIX4 的内部 8259 PIC 及 SGI 的 Visual Workstation Cobalt （IO-）APIC。但是，为了简单起见，我们在本章假定，我们的计算机是有两片 8259A PIC 的单处理机，它提供 16 个标准的 IRQ。在这种情况下，有 16 个 irq_desc_t 描述符，其中每个描述符的 handler 域指向 8259A_irq _type 变量。对其进行如下的初始化：

```
struct hw_interrupt_type i8259A_irq_type = {
          "XT-PIC",
          startup_8259A_irq,
          shutdown_8259A_irq,
          do_8259A_IRQ,
          enable_8259A_irq,
          disable_8259A_irq
    };
```

在这个结构中的第一个域“XT-PIC”是一个名字。接下来，8259A_irq_type 包含的指针指向 5 个不同的函数，这些函数就是对 PIC 编程的函数。前两个函数分别启动和关闭这个芯片的中断线。但是，在使用 8259A 芯片的情况下，这两个函数的作用与后两个函数是一样的，后两个函数是启用和禁用中断线。后面在对 do_IRQ 描述时具体描述 do_8259A_IRQ（　）函数。

3．中断服务例程描述符 irqaction

在 IRQ 描述符中我们看到指针 action 的结构为 irqaction，它是为多个设备能共享一条中断线而设置的一个数据结构。在 include/linux/interrupt.h 中定义如下:

```
  struct irqaction {
        void (*handler) (int, void *, struct pt_regs *);
         unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;
  };
```

这个描述符包含下列域。

handler

指向一个具体 I/O 设备的中断服务例程。这是允许多个设备共享同一中断线的关键域。

flags

用一组标志描述中断线与 I/O 设备之间的关系。

SA_INTERRUPT

中断处理程序必须以禁用中断来执行。

SA_SHIRQ

该设备允许其中断线与其他设备共享。

SA_SAMPLE_RANDOM

可以把这个设备看作是随机事件发生源；因此，内核可以用它做随机数产生器（用户可

以从/dev/random 和/dev/urandom 设备文件中取得随机数而访问这种特征）。

SA_PROBE

内核在执行硬件设备探测时正在使用这条中断线。

name

I/O 设备名（读取/proc/interrupts 文件，可以看到，在列出中断号时也显示设备名）。

dev_id

指定 I/O 设备的主设备号和次设备号。

next

指向 irqaction 描述符链表的下一个元素。共享同一中断线的每个硬件设备都有其对应的中断服务例程，链表中的每个元素就是对相应设备及中断服务例程的描述。

4. 中断服务例程

我们这里提到的中断服务例程（Interrupt Service Routine）与以前所提到的中断处理程序（Interrupt handler）是不同的概念。具体来说，中断处理程序相当于某个中断向量的总处理程序，例如 IRQ0x05_interrupt()，是中断号 5（向量为 37）的总处理程序，如果这个 5 号中断由网卡和图形卡共享，则网卡和图形卡分别有其相应的中断服务例程。每个中断服务例程都有相同的参数：

IRQ：中断号；

dev_id: 设备标识符，其类型为 void*；

regs: 指向内核堆栈区的指针，堆栈中存放的是中断发生后所保存的寄存器，有关 pt_regs 结构的具体内容将在后面介绍。

在实际中，大多数中断服务例程并不使用这些参数。

3.3.2 中断请求队列的初始化

在 IDT 表初始化完成之初，每个中断服务队列还为空。此时，即使打开中断且某个外设中断真的发生了，也得不到实际的服务。因为 CPU 虽然通过中断门进入了某个中断向量的总处理程序，例如 IRQ0x05_interrupt()，但是，具体的中断服务例程（如图形卡的）还没有挂入中断请求队列。因此，在设备驱动程序的初始化阶段，必须通过 request_irq()函数将对应的中断服务例程挂入中断请求队列。

request_irq()函数的代码在 / arch/i386/kernel/irq.c 中：

```
/*
 *      request_irq - allocate an interrupt line
 *      @irq: Interrupt line to allocate
 *      @handler: Function to be called when the IRQ occurs
 *      @irqflags: Interrupt type flags
 *      @devname: An ascii name for the claiming device
 *      @dev_id: A cookie passed back to the handler function
 *
 *      This call allocates interrupt resources and enables the
 *      interrupt line and IRQ handling. From the point this
 *      call is made your handler function may be invoked. Since
 *      your handler function must clear any interrupt the board
```

```
 *      raises, you must take care both to initialise your hardware
 *      and to set up the interrupt handler in the right order.
 *
 *      Dev_id must be globally unique. Normally the address of the
 *      device data structure is used as the cookie. Since the handler
 *      receives this value it makes sense to use it.
 *
 *      If your interrupt is shared you must pass a non NULL dev_id
 *      as this is required when freeing the interrupt.
 *
 *      Flags:
 *
 *      SA_SHIRQ                Interrupt is shared
 *
 *      SA_INTERRUPT            Disable local interrupts while processing
 *
 *      SA_SAMPLE_RANDOM        The interrupt can be used for entropy
 *
 */

int request_irq (unsigned int irq,
                void  (*handler) (int, void *, struct pt_regs *) ,
                unsigned long irqflags,
                const char * devname,
                void *dev_id)
{
        int retval;
        struct irqaction * action;

#if 1
        /*
        * Sanity-check: shared interrupts should REALLY pass in
        * a real dev-ID, otherwise we'll have trouble later trying
        * to figure out which interrupt is which (messes up the
        * interrupt freeing logic etc) .
        */
        if (irqflags & SA_SHIRQ) {
               if (!dev_id)
                       printk("Bad boy: %s (at 0x%x) called us without a dev_id!\n", devname,
(&irq) [-1]) ;
        }
#endif

        if (irq >= NR_IRQS)
                return -EINVAL;
        if (!handler)
               return -EINVAL;

       action = (struct irqaction *)
                        kmalloc (sizeof (struct irqaction) , GFP_KERNEL) ;
        if (!action)
               return -ENOMEM;
```

```
        action->handler = handler;
        action->flags = irqflags;
        action->mask = 0;
        action->name = devname;        /*对 action 进行初始化*/
        action->next = NULL;
        action->dev_id = dev_id;

        retval = setup_irq (irq, action) ;
        if (retval)
                kfree (action) ;
        return retval;
}
```

编码作者对此函数给出了比较详细的描述。其中主要语句就是对 setup_irq()函数的调用，该函数才是真正对中断请求队列进行初始化的函数（有所简化）：

```
int setup_irq (unsigned int irq, struct irqaction * new)
{
        int shared = 0;
        unsigned long flags;
        struct irqaction *old, **p;
        irq_desc_t *desc = irq_desc + irq;       /*获得 irq 的描述符*/

          /* 对中断请求队列的操作必须在临界区中进行 */
        spin_lock_irqsave (&desc->lock,flags) ;    /*进入临界区*/
        p = &desc->action;    /*让 p 指向 irq 描述符的 action 域，即 irqaction 链表的首部*/
       if ( (old = *p) != NULL) {   /*如果这个链表不为空*/
                /* Can't share interrupts unless both agree to */
               if (! (old->flags & new->flags & SA_SHIRQ) ) {
                       spin_unlock_irqrestore (&desc->lock,flags) ;
                     return -EBUSY;
               }

               /* 把新的中断服务例程加入到 irq 中断请求队列*/
               do {
                       p = &old->next;
                       old = *p;
               } while (old) ;
               shared = 1;
        }

        *p = new;

        if (!shared) {        /*如果 irq 不被共享 */
               desc->depth = 0;     /*启用这条 irq 线*/
               desc->status &= ~ (IRQ_DISABLED | IRQ_AUTODETECT | IRQ_WAITING) ;
               desc->handler->startup (irq) ; /*即调用 startup_8259A_irq()函数*/
        }
        spin_unlock_irqrestore (&desc->lock,flags) ;    /*退出临界区*/

        register_irq_proc (irq) ;      /*在 proc 文件系统中显示 irq 的信息*/
        return 0;
```

```
}
```

下面我们举例说明对这两个函数的使用。

1. 对 register_irq()函数的使用

在驱动程序初始化或者在设备第一次打开时，首先要调用该函数，以申请使用该 irq。其中参数 handler 指的是要挂入到中断请求队列中的中断服务例程。假定一个程序要对 / dev/fd0/（第一个软盘对应的设备）设备进行访问，有两种方式，一是直接访问 / dev/fd0/，另一种是在系统上安装一个文件系统，我们这里假定采用第一种。通常将 IRQ6 分配给软盘控制器，给定这个中断号 6，软盘驱动程序就可以发出下列请求，以将其中断服务例程挂入中断请求队列：

```
request_irq (6, floppy_interrupt,
        SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL);
```

我们可以看到，floppy_interrupt()中断服务例程运行时必须禁用中断（设置了 SA_INTERRUPT 标志），并且不允许共享这个 IRQ（清 SA_SHIRQ 标志）。

在关闭设备时，必须通过调用 free_irq()函数释放所申请的中断请求号。例如，当软盘操作终止时（或者终止对 / dev/fd0/的 I/O 操作，或者卸载这个文件系统），驱动程序就放弃这个中断号：

```
free_irq (6, NULL);
```

2. 对 setup_ irq()函数的使用

在系统初始化阶段，内核为了初始化时钟中断设备 irq0 描述符，在 time_init（　）函数中使用了下面的语句：

```
struct irqaction irq0  =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL,};
    setup_irq (0, &irq0);
```

首先，初始化类型为 irqaction 的 irq0 变量，把 handler 域设置成 timer_interrupt（　）函数的地址，flags 域设置成 SA_INTERRUPT，name 域设置成"timer"，最后一个域设置成 NULL 以表示没有用 dev_id 值。接下来，内核调用 setup_x86_irq（　），把 irq0 插入到 IRQ0 的中断请求队列：

类似地，内核初始化与 IRQ2 和 IRQ13 相关的 irqaction 描述符，并把它们插入到相应的请求队列中，在 init_IRQ（　）函数中有下面的语句：

```
struct irqaction irq2 =
        {no_action, 0, 0, "cascade", NULL,};
struct irqaction irq13 =
    {    math_error_irq, 0, 0, "fpu", NULL,};
setup_x86_irq (2, &irq2);
setup_x86_irq (13, &irq13);
```

3.4　中断处理

通过上面的介绍，我们知道了中断描述符表已被初始化，并具有了相应的内容；对于外

部中断，中断请求队列也已建立。但是，中断处理程序并没有执行，如何执行中断处理程序正是我们本节要关心的主要内容

3.4.1 中断和异常处理的硬件处理。

首先，我们从硬件的角度来看 CPU 如何处理中断和异常。这里假定内核已被初始化，CPU 已从实模式转到保护模式。

当 CPU 执行了当前指令之后，CS 和 EIP 这对寄存器中所包含的内容就是下一条将要执行指令的逻辑地址。在对下一条指令执行前，CPU 先要判断在执行当前指令的过程中是否发生了中断或异常。如果发生了一个中断或异常，那么 CPU 将做以下事情。

- 确定所发生中断或异常的向量 *i*（在 0～255 之间）。
- 通过 IDTR 寄存器找到 IDT 表，读取 IDT 表第 *i* 项（或叫第 *i* 个门）。
- 分两步进行有效性检查：首先是“段”级检查，将 CPU 的当前特权级 CPL（存放在 CS 寄存器的最低两位）与 IDT 中第 *i* 项段选择符中的 DPL 相比较，如果 DPL（3）大于 CPL（0），就产生一个“通用保护”异常（中断向量 13），因为中断处理程序的特权级不能低于引起中断的程序的特权级。这种情况发生的可能性不大，因为中断处理程序一般运行在内核态，其特权级为 0。然后是“门”级检查，把 CPL 与 IDT 中第 *i* 个门的 DPL 相比较，如果 CPL 大于 DPL，也就是当前特权级（3）小于这个门的特权级（0），CPU 就不能“穿过”这个门，于是产生一个“通用保护”异常，这是为了避免用户应用程序访问特殊的陷阱门或中断门。但是请注意，这种“门”级检查是针对一般的用户程序，而不包括外部 I/O 产生的中断或因 CPU 内部异常而产生的异常，也就是说，如果产生了中断或异常，就免去了“门”级检查。
- 检查是否发生了特权级的变化。当中断发生在用户态（特权级为 3），而中断处理程序运行在内核态（特权级为 0），特权级发生了变化，所以会引起堆栈的更换。也就是说，从用户堆栈切换到内核堆栈。而当中断发生在内核态时，即 CPU 在内核中运行时，则不会更换堆栈，如图 3.6 所示。

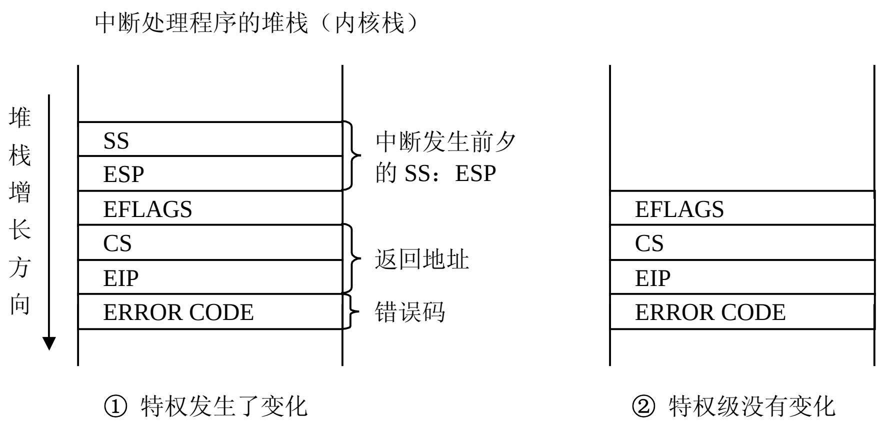


图 3.6 中断处理程序堆栈示意图

从图 3.5 中可以看出，当从用户态堆栈切换到内核态堆栈时，先把用户态堆栈的值压入中

断程序的内核态堆栈中，同时把 EFLAGS 寄存器自动压栈，然后把被中断进程的返回地址压入堆栈。如果异常产生了一个硬件错误码，则将它也保存在堆栈中。如果特权级没有发生变化，则压入栈中的内容如图 3.6 中②。你可能要问，现在 SS:ESP 和 CS:EIP 这两对寄存器的值分别是什么？SS:ESP 的值从当前进程的 TSS 中获得，也就是获得当前进程的内核栈指针，因为此时中断处理程序成为当前进程的一部分，代表当前进程在运行。CS:EIP 的值就是 IDT 表中第 i 项门描述符的段选择符和偏移量的值，此时，CPU 就跳转到了中断或异常处理程序。

3.4.2　Linux 对中断的处理

上面给出了硬件对异常和中断进行处理的一般步骤，下面将概要描述 Linux 对中断的处理，具体的实现过程将在后面介绍。

当一个中断发生时，并不是所有的操作都具有相同的急迫性。事实上，把所有的操作都放进中断处理程序本身并不合适。需要时间长的、非重要的操作应该推后，因为当一个中断处理程序正在运行时，相应的 IRQ 中断线上再发出的信号就会被忽略。更重要的是，中断处理程序是代表进程执行的，它所代表的进程必须总处于 TASK_RUNNING 状态，否则，就可能出现系统僵死情形。因此，中断处理程序不能执行任何阻塞过程，如 I/O 设备操作。因此，Linux 把一个中断要执行的操作分为下面的 3 类。

1．紧急的（Critical）

这样的操作诸如：中断到来时中断控制器做出应答，对中断控制器或设备控制器重新编程，或者对设备和处理器同时访问的数据结构进行修改。这些操作都是紧急的，应该被很快地执行，也就是说，紧急操作应该在一个中断处理程序内立即执行，而且是在禁用中断的状态下。

2．非紧急的（Noncritical）

这样的操作如修改那些只有处理器才会访问的数据结构（例如，按下一个键后，读扫描码）。这些操作也要很快地完成，因此，它们由中断处理程序立即执行，但在启用中断的状态下。

3．非紧急可延迟的（Noncritical deferrable）

这样的操作如，把一个缓冲区的内容拷贝到一些进程的地址空间（例如，把键盘行缓冲区的内容发送到终端处理程序的进程）。这些操作可能被延迟较长的时间间隔而不影响内核操作：有兴趣的进程会等待需要的数据。非紧急可延迟的操作由一些被称为“下半部分”（bottom halves）的函数来执行。我们将在后面讨论“下半部分”。

所有的中断处理程序都执行 4 个基本的操作：

- 在内核栈中保存 IRQ 的值和寄存器的内容；
- 给与 IRQ 中断线相连的中断控制器发送一个应答，这将允许在这条中断线上进一步发出中断请求；
- 执行共享这个 IRQ 的所有设备的中断服务例程（ISR）；
- 跳到 ret_from_intr（　）的地址后终止。

3.4.3　与堆栈有关的常量、数据结构及宏

在中断、异常及系统调用的处理中，涉及一些相关的常量、数据结构及宏，在此先给予介绍（大部分代码在 arch/i386/kernel/entry.S 中）。

1．常量定义

下面这些常量定义了进入中断处理程序时，相关寄存器与堆栈指针（ESP）的相对位置，图 3.7 给出了在相应位置上所保存的寄存器内容。

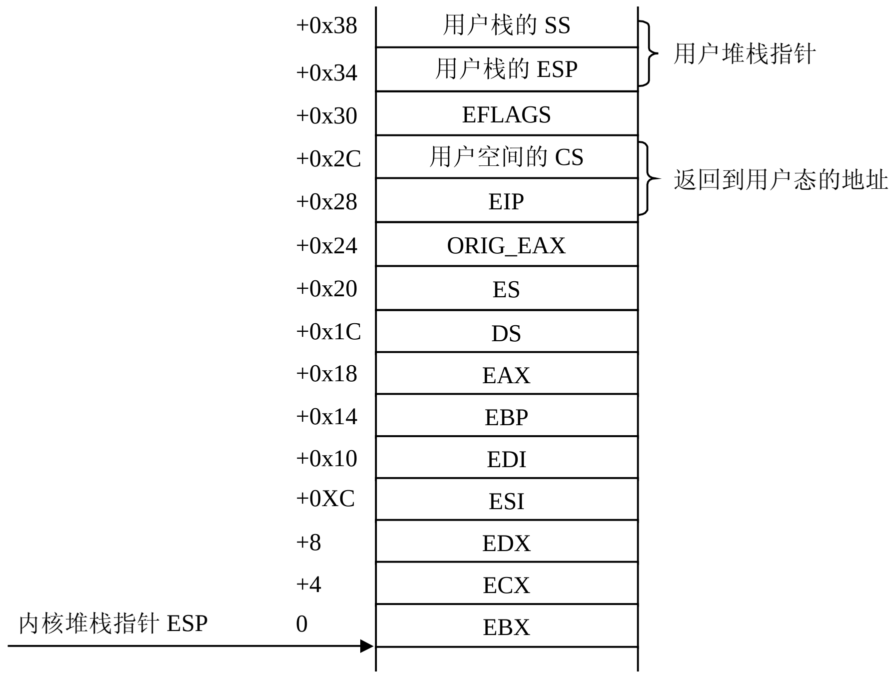


图 3.7　进入中断理程序时内核堆栈示意图

```
EBX = 0x00
ECX= 0x04
EDX= 0x08
ESI= 0x0C
EDI= 0x10
EB = 0x14
EAX= 0x18
DS= 0x1C
ES = 0x20
ORIG_EAX = 0x24
EIP = 0x28
CS  = 0x2C
EFLAGS = 0x30
OLDESP= 0x34
OLDSS = 0x38
```

其中，ORIG_EAX 是 Original eax 之意，其具体含义将在后面介绍。

2. 存放在栈中的寄存器结构 pt_regs

在内核中，很多函数的参数是 pt_regs 数据结构，定义在 include/i386/ptrace.h 中：

```
   struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

把这个结构与内核栈的内容相比较，会发现堆栈的内容是这个数据结构的一个映像。

3. 保存现场的宏 SAVE_ALL

在中断发生前夕，要把所有相关寄存器的内容都保存在堆栈中,这是通过 SAVE_ALL 宏完成的：

```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $ (__KERNEL_DS) ,%edx; \
        movl %edx,%ds; \
        movl %edx,%es;
```

该宏执行以后，堆栈内容如图 3.7 所示。把这个宏与图 3.6 结合起来就很容易理解图 3.7，在此对该宏再给予解释：

- CPU 在进入中断处理程序时自动将用户栈指针（如果更换堆栈）、EFLAGS 寄存器及返回地址一同压入堆栈。
- 段寄存器 DS 和 ES 原来的内容入栈，然后装入内核数据段描述符__KERNEL_DS（定义为 0x18），内核段的 DPL 为 0。

4．恢复现场的宏 RESTORE_ALL

当从中断返回时，恢复相关寄存器的内容，这是通过 RESTORE_ALL 宏完成的：

```
    #define RESTORE_ALL      \
        popl %ebx;       \
        popl %ecx;       \
         popl %edx;       \
         popl %esi;       \
         popl %edi;       \
         popl %ebp;       \
         popl %eax;       \
  1:      popl %ds;        \
  2:      popl %es;        \
          addl $4,%esp;    \
  3:      iret;
```

可以看出，RESTORE_ALL 与 SAVE_ALL 遥相呼应。当执行到 iret 指令时，内核栈又恢复到刚进入中断门时的状态，并使 CPU 从中断返回。

5．将当前进程的 task_struct 结构的地址放在寄存器中

```
 #define GET_CURRENT（reg） \
         movl $-8192, reg; \
         andl %esp, reg
```

从下一章“task_struct 结构在内存存放”一节我们将知道，当前进程的 task_struct 存放在内核栈的底部，因此，以上两条指令就可以把 task_struct 结构的地址放在 reg 寄存器中。

3.4.4 中断处理程序的执行

从前面的介绍，我们已经知道了 i386 的中断机制及有关的初始化工作。现在，我们可以从中断请求的发生到 CPU 的响应，再到中断处理程序的调用和返回，沿着这一思路走一遍，以体会 Linux 内核对中断的响应及处理。

假定外设的驱动程序都已完成了初始化工作，并且已把相应的中断服务例程挂入到特定的中断请求队列。又假定当前进程正在用户空间运行（随时可以接受中断），且外设已产生了一次中断请求。当这个中断请求通过中断控制器 8259A 到达 CPU 的中断请求引线 INTR 时（参看图 3.1），CPU 就在执行完当前指令后来响应该中断。

CPU 从中断控制器的一个端口取得中断向量 I，然后根据 I 从中断描述符表 IDT 中找到相应的表项，也就是找到相应的中断门。因为这是外部中断，不需要进行“门级”检查，CPU 就可以从这个中断门获得中断处理程序的入口地址，假定为 IRQ0x05_interrupt。因为这里假定中断发生时 CPU 运行在用户空间（CPL＝3），而中断处理程序属于内核（DPL＝0），因此，要进行堆栈的切换。也就是说，CPU 从 TSS 中取出内核栈指针，并切换到内核栈（此时栈还为空）。当 CPU 进入 IRQ0x05_interrupt 时，内核栈如图 3.6 的①，栈中除用户栈指针、EFLAGS 的内容以及返回地址外再无其他内容。另外，由于 CPU 进入的是中断门（而不是陷阱门），因此，这条中断线已被禁用，直到重新启用。

我们用 IRQn_interrupt 来表示从 IRQ0x01_interrupt 到 IRQ0x0f_interrupt 任意一个中断处理程序。这个中断处理程序实际上要调用 do_IRQ()，而 do_IRQ()要调用 handle_IRQ_event()函数；最后这个函数才真正地执行中断服务例程（ISR）。图 3.8 给出了它们的调用关系。

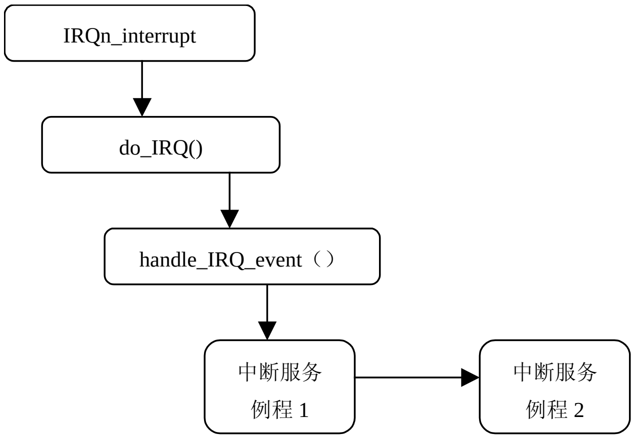


图 3.8　中断处理函数的调用关系

1．中断处理程序 IRQn_interrupt

我们首先看一下从 IRQ0x01_interrupt 到 IRQ0x0f_interrupt 的这 16 个函数是如何定义的，在 i8259.c 中定义了如下宏：

```
#define BI（x,y） \
        BUILD_IRQ（x##y）

#define BUILD_16_IRQS（x） \
        BI（x,0） BI（x,1） BI（x,2） BI（x,3） \
        BI（x,4） BI（x,5） BI（x,6） BI（x,7） \
        BI（x,8） BI（x,9） BI（x,a） BI（x,b） \
        BI（x,c） BI（x,d） BI（x,e） BI（x,f）

BUILD_16_IRQS（0x0）
```

经过 gcc 的预处理，宏定义 BUILD_16_IRQS（0x0） 会被展开成 BUILD_IRQ（0x00）至 BUILD_IRQ（0x0f）。BUILD_IRQ 宏是一段嵌入式汇编代码（在/include/i386/hw_irq.h 中），为了有助于理解，我们把它展开成下面的汇编语言片段：

```
IRQn_interrupt:
        pushl $n-256
jmp common_interrupt
```

把中断号减 256 的结果保存在栈中，这就是进入中断处理程序后第一个压入堆栈的值，也就是堆栈中 ORIG_EAX 的值，如图 3.7。这是一个负数，正数留给系统调用使用。对于每个中断处理程序，唯一不同的就是压入栈中的这个数。然后，所有的中断处理程序都跳到一段相同的代码 common_interrupt。这段代码可以在 BUILD_COMMON_IRQ 宏中找到，同样，我们略去其嵌入式汇编源代码，而把这个宏展开成下列的汇编语言片段：

```
common_interrupt:
        SAVE_ALL
```

```
        call do_IRQ
    jmp ret_from_intr
```

SAVE_ALL 宏已经在前面介绍过，它把中断处理程序会使用的所有 CPU 寄存器都保存在栈中。然后，BUILD_COMMON_IRQ 宏调用 do_IRQ（　）函数，因为通过 CALL 调用这个函数，因此，该函数的返回地址被压入栈。当执行完 do_IRQ（　），就跳转到 ret_from_intr（　）地址（参见后面的“从中断和异常返回”）。

2．do_IRQ（　）函数

do_IRQ()这个函数处理所有外设的中断请求。当这个函数执行时，内核栈从栈顶到栈底包括：

- do_IRQ（　）的返回地址；
- 由 SAVE_ALL 推进栈中的一组寄存器的值；
- ORIG_EAX（即 *n*-256）；
- CPU 自动保存的寄存器。

该函数的实现用到中断线的状态，下面给予具体说明：

```
#define IRQ_INPROGRESS  1   /* 正在执行这个 IRQ 的一个处理程序*/
#define IRQ_DISABLED    2    /* 由设备驱动程序已经禁用了这条 IRQ 中断线 */
#define IRQ_PENDING     4    /* 一个 IRQ 已经出现在中断线上，且被应答，但还没有为它提供服务 */
#define IRQ_REPLAY      8    /* 当 Linux 重新发送一个已被删除的 IRQ 时 */
#define IRQ_AUTODETECT  16   /* 当进行硬件设备探测时，内核使用这条 IRQ 中断线 */
#define IRQ_WAITING     32   /*当对硬件设备进行探测时，设置这个状态以标记正在被测试的 irq */
#define IRQ_LEVEL       64    /* IRQ level triggered */
#define IRQ_MASKED      128    /* IRQ masked - shouldn't be seen again */
#define IRQ_PER_CPU     256     /* IRQ is per CPU */
```

这 9 个状态的前 6 个状态比较常用，因此我们给出了具体解释。另外，我们还看到每个状态的常量是 2 的幂次方。最大值为 256（2^8），因此可以用一个字节来表示这 9 个状态，其中每一位对应一个状态。

该函数在 arch//i386/kernel/irq.c 中定义如下：

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
        /*  函数返回 0 则意味着这个 irq 正在由另一个 CPU 进行处理，
或这条中断线被禁用*/
        int irq = regs.orig_eax & 0xff;      /* 还原中断号 */
        int cpu = smp_processor_id();     /*获得 CPU 号*/
        irq_desc_t *desc = irq_desc + irq; /*在 irq_desc[]数组中获得 irq 的描述符*/
        struct irqaction * action;
        unsigned int status;

        kstat.irqs[cpu][irq]++;
```

```
        spin_lock（&desc->lock）；  /＊针对多处理机加锁*/
        desc->handler->ack（irq）；  /＊CPU 对中断请求给予确认*/

        status = desc->status & ~（IRQ_REPLAY | IRQ_WAITING）；
        status |= IRQ_PENDING; /* we _want_ to handle it */


        action = NULL;
        if （!（status & （IRQ_DISABLED | IRQ_INPROGRESS）））{
                action = desc->action;
                status &= ~IRQ_PENDING; /* we commit to handling */
                status |= IRQ_INPROGRESS; /* we are handling it */
        }
        desc->status = status;
        if （!action）
                goto out;
        for （;;）{
                 spin_unlock（&desc->lock）；  /*进入临界区*
                 handle_IRQ_event（irq，&regs，action）；
                 spin_lock（&desc->lock）；   /*出临界区*/

                 if （!（desc->status & IRQ_PENDING））
                         break;
                 desc->status &= ~IRQ_PENDING;
        }
         desc->status &= ~IRQ_INPROGRESS;
out:
         /*
          * The ->end() handler has to deal with interrupts which got
         * disabled while the handler was running.
          */
         desc->handler->end（irq）；
         spin_unlock（&desc->lock）；

         if （softirq_pending（cpu））
                 do_softirq();   /*处理软中断*/
         return 1;
}
```

下面对这个函数进行进一步的讨论。

- 当执行到 for （;;）这个无限循环时，就准备对中断请求队列进行处理，这是由 handle_IRQ_event ()函数完成的。因为中断请求队列为一临界资源，因此在进入这个函数前要加锁。
- handle_IRQ_event ()函数的主要代码片段为：

```
if （!（action->flags & SA_INTERRUPT））
                __sti();   /*关中断*/
do {
               status |= action->flags;
                 action->handler（irq，action->dev_id，regs）；
                 action = action->next;
         } while （action）；
```

```
__cli();     /*开中断*/
```

这个循环依次调用请求队列中的每个中断服务例程。中断服务例程及其参数已经在前面进行过简单描述，至于更具体的解释将在驱动程序一章进行描述。

- 这里要说明的是，中断服务例程都在关中断的条件下进行（不包括非屏蔽中断），这也是为什么 CPU 在穿过中断门时自动关闭中断的原因。但是，关中断时间绝不能太长，否则就可能丢失其他重要的中断。也就是说，中断服务例程应该处理最紧急的事情，而把剩下的事情交给另外一部分来处理。即后半部分（bottom half）来处理，这一部分内容将在下一节进行讨论。
- 经验表明，应该避免在同一条中断线上的中断嵌套，内核通过 IRQ_PENDING 标志位的应用保证了这一点。当 do_IRQ()执行到 for（;;）循环时，desc->status 中的 IRQ_PENDING 的标志位肯定为 0（想想为什么？）。当 CPU 执行完 handle_IRQ_event ()函数返回时，如果这个标志位仍然为 0，那么循环就此结束。如果这个标志位变为 1，那就说明这条中断线上又有中断产生（对单 CPU 而言），所以循环又执行一次。通过这种循环方式，就把可能发生在同一中断线上的嵌套循环化解为“串行”。
- 不同的 CPU 不允许并发地进入同一中断服务例程，否则，那就要求所有的中断服务例程必须是“可重入”的纯代码。可重入代码的设计和实现就复杂多了，因此，Linux 在设计内核时巧妙地“避难就易”，以解决问题为主要目标。
- 在循环结束后调用 desc->handler->end()函数，具体来说，如果没有设置 IRQ_DISABLED 标志位，就调用低级函数 enable_8259A_irq()来启用这条中断线。
- 如果这个中断有后半部分,就调用 do_softirq()执行后半部分。

3.4.5　从中断返回

从前面的讨论我们知道，do_IRQ()这个函数处理所有外设的中断请求。这个函数执行的时候，内核栈栈顶包含的就是 do_IRQ()的返回地址，这个地址指向 ret_from_intr。实际上，ret_from_intr 是一段汇编语言的入口点，为了描述简单起见，我们以函数的形式提及它。虽然我们这里讨论的是中断的返回，但实际上中断、异常及系统调用的返回是放在一起实现的，因此，我们常常以函数的形式提到下面这 3 个入口点。

ret_from_intr()

终止中断处理程序。

ret_from_sys_call（ ）

终止系统调用，即由 0x80 引起的异常。

ret_from_exception（ ）

终止除了 0x80 的所有异常。

在相关的计算机课程中，我们已经知道从中断返回时 CPU 要做的事情，下面我们来看一下 Linux 内核的具体实现代码（在 entry.S 中）：

```
ENTRY（ret_from_intr）
        GET_CURRENT（%ebx）
ret_from_exception:
         movl EFLAGS（%esp）,%eax          # mix EFLAGS and CS
         movb CS（%esp）,%al
```

```
        testl $（VM_MASK | 3）,%eax       # return to VM86 mode or non-supervisor?
        jne ret_from_sys_call
        jmp restore_all
```

这里的 GET_CURRENT（%ebx）将当前进程 task_struct 结构的指针放入寄存器 EBX 中，此时，内核栈的内容还如图 3.7 所示。然后两条“mov”指令是为了把中断发生前夕 EFALGS 寄存器的高 16 位与代码段 CS 寄存器的内容拼揍成 32 位的长整数，其目的是要检验：

- 中断前夕 CPU 是否够运行于 VM86 模式；
- 中断前夕 CPU 是运行在用户空间还是内核空间。

VM86 模式是为在 i386 保护模式下模拟运行 DOS 软件而设置的，EFALGS 寄存器高 16 位中有个标志位表示 CPU 是否运行在 VM86 模式，我们在此不予详细讨论。CS 的最低两位表示中断发生时 CPU 的运行级别 CPL，若这两位为 3，说明中断发生于用户空间。

如果中断发生在内核空间，则控制权直接转移到 restore_all。如果中断发生于用户空间（或 VM86 模式），则转移到 ret_from_sys_call：

```
ENTRY（ret_from_sys_call）
        cli                            # need_resched and signals atomic test
        cmpl $0,need_resched（%ebx）
        jne reschedule
        cmpl $0,sigpending（%ebx）
        jne signal_return
restore_all:
        RESTORE_ALL

  reschedule:
        call SYMBOL_NAME（schedule）    # test
        jmp ret_from_sys_call
```

进入 ret_from_sys_call 后，首先关中断，也就是说，执行这段代码时 CPU 不接受任何中断请求。然后，看调度标志是否为非 0，其中常量 need_resched 定义为 20，need_resched（%ebx）表示当前进程 task_struct 结构中偏移量 need_resched 处的内容，如果调度标志为非 0，说明需要进行调度，则去调用 schedule()函数进行进程调度，这将在第五章进行详细讨论。

同样，如果当前进程的 task_struct 结构中的 sigpending 标志为非 0，则表示该进程有信号等待处理，要先处理完这些信号后才从中断返回，关于信号的处理将在“进程间通信”一章进行讨论。处理完信号，控制权还是返回到 restore_all。RESTORE_ALL 宏操作在前面已经介绍过，也就是恢复中断现场，彻底从中断返回。

3.5　中断的后半部分处理机制

从上面的讨论我们知道，Linux 并不是一口气把中断所要求的事情全部做完，而是分两部分来做，本节我们将具体描述内核怎样处理中断的后半部分。

3.5.1 为什么把中断分为两部分来处理

中断服务例程一般都是在中断请求关闭的条件下执行的,以避免嵌套而使中断控制复杂化。但是，中断是一个随机事件，它随时会到来，如果关中断的时间太长，CPU 就不能及时响应其他的中断请求，从而造成中断的丢失。因此，内核的目标就是尽可能快地处理完中断请求，尽其所能把更多的处理向后推迟。例如，假设一个数据块已经达到了网线，当中断控制器接受到这个中断请求信号时，Linux 内核只是简单地标志数据到来了，然后让处理器恢复到它以前运行的状态，其余的处理稍后再进行（如把数据移入一个缓冲区，接受数据的进程就可以在缓冲区找到数据）。因此，内核把中断处理分为两部分：前半部分（top half）和后半部分（bottom half），前半部分内核立即执行，而后半部分留着稍后处理，如图 3.9 所示。

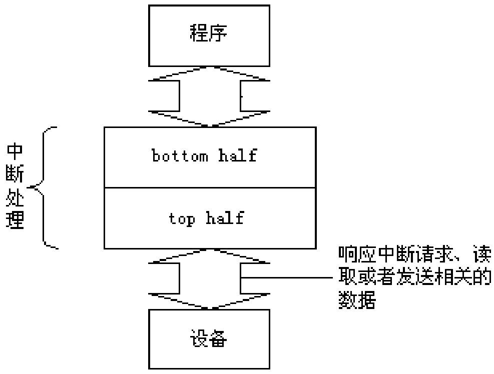


图 3.9 中断的分割

首先，一个快速的“前半部分”来处理硬件发出的请求，它必须在一个新的中断产生之前终止。通常情况下，除了在设备和一些内存缓冲区（如果设备用到了 DMA，就不止这些）之间移动或传送数据，确定硬件是否处于健全的状态之外，这一部分做的工作很少。

然后，就让一些与中断处理相关的有限个函数作为“后半部分”来运行：

- 允许一个普通的内核函数，而不仅仅是服务于中断的一个函数，能以后半部分的身份来运行；
- 允许几个内核函数合在一起作为一个后半部分来运行。

后半部分运行时是允许中断请求的，而前半部分运行时是关中断的，这是二者之间的主要区别。

3.5.2 实现机制

Linux 内核为将中断服务分为两部分提供了方便，并设立了相应的机制。在以前的内核中，这个机制就叫 bottom half（简称 bh），但在 2.4 版中有了新的发展和推广，叫做软中断（softirq）机制。

1．bh 机制

以前内核中的 bh 机制设置了一个函数指针数组 bh_base[]，它把所有的后半部分都组织起来，其大小为 32，数组中的每一项就是一个后半部分，即一个 bh 函数。同时，又设置了两个 32 位无符号整数 bh_active 和 bh_mask，每个无符号整数中的一位对应着 bh_base[]中的一个元素，如图 3.10 所示。

在 2.4 以前的内核中，每次执行完 do_IRQ()中的中断服务例程以后，以及每次系统调用结束之前，就在一个叫 do_bottom_half()的函数中执行相应的 bh 函数。

在 do_bottom_half()中对 bh 函数的执行是在关中断的情况下进行的，也就是说对 bh 的执行进行了严格的“串行化”，这种方式简化了 bh 的设计，这是因为，对单 CPU 来说，bh 函数的执行可以不嵌套；而对于多 CPU 来说，在同一时间内最多只允许一个 CPU 执行 bh 函数。

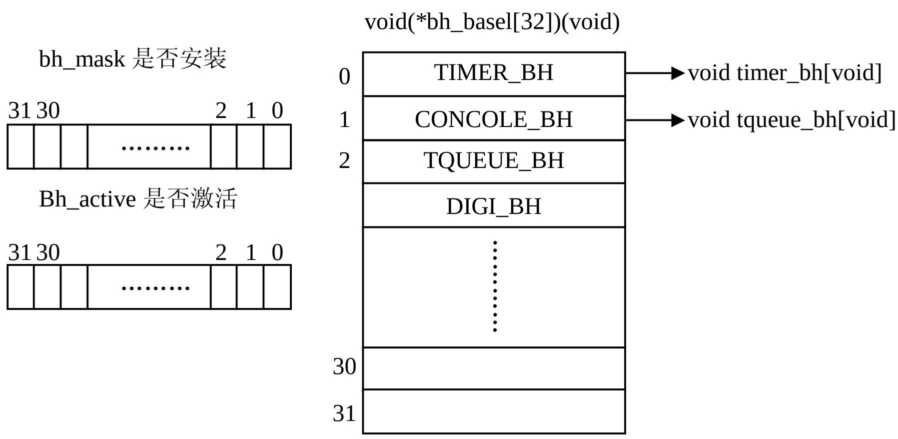


图 3.10　bh 机制示意图

这种简化了的设计在一定程度上保证了从单 CPU 到多 CPU SMP 结构的平稳过渡，但随着时间的推移，就会发现这样的处理对于 SMP 的性能有不利的影响。因为，当系统中有很多个 bh 函数需要执行时，bh 函数的“串行化”却只能使一个 CPU 执行一个 bh 函数，其他 CPU 即使空闲，也不能执行其他的 bh 函数。由此可以看出，bh 函数的串行化是针对所有 CPU 的，根本发挥不出多 CPU 的优势。

那么，在新内核的设计中，是改进 bh 机制还是抛弃 bh 机制，建立一种新的机制？2.4 选择了一种折衷的办法，继续保留 bh 机制，另外增加一种或几种机制，并把它们纳入一个统一的框架中，这就是 2.4 内核中的软中断（softirq）机制。

2．软中断机制

软中断机制也是推迟内核函数的执行，然而，与 bh 函数严格地串行执行相比，软中断却在任何时候都不需要串行化。同一个软中断的两个实例完全有可能在两个 CPU 上同时运行。当然，在这种情况下，软中断必须是可重入的。软中断给网络部分带来的好处尤为突出，因为 2.4 内核中用两个软中断代替原来的一个 NET_BH 函数，这就使得在多处理机系统上软中断的执行更为高效。

3．Tasklet 机制

另一个类似于 bh 的机制叫做 tasklet。Tasklet 建立在软中断之上，但与软中断的区别是，同一个 tasklet 只能运行在一个 CPU 上，而不同的 tasklet 可以同时运行在不同的 CPU 上。在这种情况下，tasklet 就不需要是可重入的，因此，编写 tasklet 比编写一个软中断要容易。

Bh 机制在 2.4 中依然存在，但不是作为一个单独的机制存在，而是建立在 tasklet 之上。因此，在 2.4 版中，设备驱动程序的开发者必须更新他们原来的驱动程序，用 tasklet 代替 bh。

3.5.3　数据结构的定义

在具体介绍软中断处理机制之前，我们先介绍一下相关的数据结构，这些数据结构大部分都在 / include/linux/interrupt.h 中。

1．与软中断相关的数据结构

软中断本身是一种机制，同时也是一种基本框架。在这个框架中，既包含了 bh 机制，也包含了 tasklet 机制。

（1）内核定义的软中断

```
enum
        {
         HI_SOFTIRQ=0,
         NET_TX_SOFTIRQ,
         NET_RX_SOFTIRQ,
         TASKLET_SOFTIRQ
         };
```

内核中用枚举类型定义了 4 种类型的软中断，其中 NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ 两个软中断是专为网络操作而设计的，而 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 是针对 bh 和 tasklet 而设计的软中断。编码作者在源码注释中曾提到，一般情况下，不要再分配新的软中断。

（2）软中断向量

```
      struct softirq_action
{
                void    (*action)（struct softirq_action *）;
      void   *data;
         }

     static struct softirq_action softirq_vec[32] __cacheline_aligned;
```

从定义可以看出，内核定义了 32 个软中断向量，每个向量指向一个函数，但实际上，内核目前只定义了上面的 4 个软中断，而我们后面主要用到的为 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 两个软中断。

（3）软中断控制 / 状态结构

softirq_vec [] 是个全局量，系统中每个 CPU 所看到的是同一个数组。但是，每个 CPU

各有其自己的“软中断控制 / 状态”结构，这些数据结构形成一个以 CPU 编号为下标的数组 irq_stat[]（定义在 include/i386/hardirq.h 中）

```
typedef struct {
        unsigned int __softirq_pending;
        unsigned int __local_irq_count;
        unsigned int __local_bh_count;
        unsigned int __syscall_count;
        struct task_struct * __ksoftirqd_task; /* waitqueue is too large */
        unsigned int __nmi_count;      /* arch dependent */
} ____cacheline_aligned irq_cpustat_t;

irq_cpustat_t irq_stat[NR_CPUS];
```

irq_stat[]数组也是一个全局量，但是各个 CPU 可以按其自身的编号访问相应的域。于是，内核定义了如下宏（在 include/linux/irq_cpustat.h 中）：

```
#ifdef CONFIG_SMP
#define __IRQ_STAT（cpu, member）  （irq_stat[cpu].member）
#else
#define __IRQ_STAT（cpu, member）  （（void）（cpu）, irq_stat[0].member）
#endif

/* arch independent irq_stat fields */
#define softirq_pending（cpu）    __IRQ_STAT（（cpu）, __softirq_pending）
#define local_irq_count（cpu）    __IRQ_STAT（（cpu）, __local_irq_count）
#define local_bh_count（cpu）     __IRQ_STAT（（cpu）, __local_bh_count）
#define syscall_count（cpu）      __IRQ_STAT（（cpu）, __syscall_count）
#define ksoftirqd_task（cpu）     __IRQ_STAT（（cpu）, __ksoftirqd_task）
   /* arch dependent irq_stat fields */
#define nmi_count（cpu）          __IRQ_STAT（（cpu）, __nmi_count）          /* i386, ia64
*/
```

2. 与 tasklet 相关的数据结构

与 bh 函数相比，tasklet 是“多序”的 bh 函数。内核中用 tasklet_task 来定义一个 tasklet：

```
 struct tasklet_struct
 {
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void （*func）（unsigned long）;
        unsigned long data;
};
```

从定义可以看出，tasklet_struct 是一个链表结构，结构中的函数指针 func 指向其服务程序。内核中还定义了一个以 CPU 编号为下标的数组 tasklet_vec[]和 tasklet_hi_vec[]：

```
  struct tasklet_head
{
        struct tasklet_struct *list;
} __attribute__ （（__aligned__（SMP_CACHE_BYTES）））;
```

```
extern struct tasklet_head tasklet_vec[NR_CPUS];
extern struct tasklet_head tasklet_hi_vec[NR_CPUS];
```

这两个数组都是 tasklet_head 结构数组，每个 tasklet_head 结构就是一个 tasklet_struct 结构的队列头。

3．与 bh 相关的数据结构

前面我们提到，bh 建立在 tasklet 之上，更具体地说，对一个 bh 的描述也是 tasklet_struct 结构，只不过执行机制有所不同。因为在不同的 CPU 上可以同时执行不同的 tasklet，而任何时刻，即使在多个 CPU 上，也只能有一个 bh 函数执行。

（1）bh 的类型

```
enum {
        TIMER_BH = 0,  /* 定时器 */
        TQUEUE_BH,     /* 周期性任务队列 */
        DIGI_BH,       /* DigiBoard PC/Xe */
        SERIAL_BH,     /* 串行接口 */
        RISCOM8_BH,        /* RISCom/8 */
        SPECIALIX_BH,  /* Specialix IO8+ */
        AURORA_BH,         /* Aurora 多端口卡（SPARC）*/
        ESP_BH,            /* Hayes ESP 串行卡 */
        SCSI_BH,       /* SCSI 接口*/
        IMMEDIATE_BH,  /* 立即任务队列*/
        CYCLADES_BH,   /* Cyclades Cyclom-Y 串行多端口 */
        CM206_BH,      /* CD-ROM Philips/LMS cm206 磁盘 */
        JS_BH,             /* 游戏杆（PC IBM）*/
        MACSERIAL_BH,  /* Power Macintosh 的串行端口 */
        ISICOM_BH      /* MultiTech 的 ISI 卡*/
};
```

在给出 bh 定义的同时，我们也给出了解释。从定义中可以看出，有些 bh 与硬件设备相关，但这些硬件设备未必装在系统中，或者仅仅是针对 IBM PC 兼容机之外的某些平台。

（2）bh 的组织结构

在 2.4 以前的版本中，把所有的 bh 用一个 bh_base[]数组组织在一起，数组的每个元素指向一个 bh 函数：

```
static void （*bh_base[32]）（void）;
```

2.4 版中保留了上面这种定义形式，但又定义了另外一种形式：

```
struct tasklet_struct bh_task_vec[32];
```

这也是一个有 32 个元素的数组，但数组的每个元素是一个 tasklet_struct 结构，数组的下标就是上面定义的枚举类型中的序号。

3.5.4 软中断、bh 及 tasklet 的初始化

1．Tasklet 的初始化

Tasklet 的初始化是由 tasklet_ init()函数完成的：

```
void tasklet_init（struct tasklet_struct *t,
```

```
                    void （*func）（unsigned long），unsigned long data）
{
         t->next = NULL;
         t->state = 0;
         atomic_set（&t->count, 0）;
         t->func = func;
         t->data = data;
}
```

其中，atomic_set()为原子操作,它把 t->count 置为 0。

2．软中断的初始化

首先通过 open_softirq()函数打开软中断：

```
   void open_softirq（int nr, void （*action）（struct softirq_action*），void *data）
{
         softirq_vec[nr].data = data;
         softirq_vec[nr].action = action;
 }
```

然后，通过 softirq_init()函数对软中断进行初始化：

```
void __init softirq_init()
{
         int i;

         for （i=0; i<32; i++）
                tasklet_init（bh_task_vec+i, bh_action, i）;

         open_softirq（TASKLET_SOFTIRQ, tasklet_action, NULL）;
         open_softirq（HI_SOFTIRQ, tasklet_hi_action, NULL）;
}
```

对于 bh 的 32 个 tasklet_struct，调用 tasklet_init 以后，它们的函数指针 func 全部指向 bh_action()函数，也就是建立了 bh 的执行机制，但具体的 bh 函数还没有与之挂勾，就像具体的中断服务例程还没有挂入中断服务队列一样。同样，调用 open_softirq()以后，软中断 TASKLET_SOFTIRQ 的服务例程为 tasklet_action()，而软中断 HI_SOFTIRQ 的服务例程为 tasklet_hi_action()。

3．Bh 的初始化

bh 的初始化是由 init_bh()完成的：

```
void init_bh（int nr, void （*routine）（void））
{
         bh_base[nr] = routine;
         mb();
 }
```

这里调用的函数 mb()与 CPU 中执行指令的流水线有关，我们对此不进行进一步讨论。下面看一下几个具体 bh 的初始化（在 kernel/sched.c 中）：

```
init_bh（TIMER_BH,timer_bh）;
init_bh（TUEUE_BH,tqueue_bh）;
init_bh（IMMEDIATE_BH,immediate_bh）;
```

初始化以后，bh_base[TIMER_BH]处理定时器队列 timer_bh，每个时钟中断都会激活 TIMER_BH,在第五章将会看到,这意味着大约每隔 10ms 这个队列运行一次。bh_base[TUEUE_BH]处理周期性的任务队列 tqueue_bh，而 bh_base[IMMEDIATE_BH]通常被驱动程序所调用，请求某个设备服务的内核函数可以链接到 IMMEDIATE_BH 所管理的队列 immediate_bh 中，在该队列中排队等待。

3.5.5 后半部分的执行

1．Bh 的处理

当需要执行一个特定的 bh 函数（例如 bh_base[TIMER_BH]()）时，首先要提出请求，这是由 mark_bh()函数完成的（在 Interrupt.h 中）：

```
static inline void mark_bh（int nr）
{
        tasklet_hi_schedule（bh_task_vec+nr）;
}
```

从上面的介绍我们已经知道，bh_task_vec[]每个元素为 tasklet_struct 结构，函数的指针 func 指向 bh_action()。

接下来，我们来看 tasklet_hi_schedule()完成什么功能：

```
static inline void tasklet_hi_schedule（struct tasklet_struct *t）
{
        if （!test_and_set_bit（TASKLET_STATE_SCHED, &t->state））
        int cpu = smp_processor_id();
        unsigned long flags;

       local_irq_save（flags）;
        t->next = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = t;
        cpu_raise_softirq（cpu, HI_SOFTIRQ）;
        local_irq_restore（flags）;
}
```

其中 smp_processor_id()返回当前进程所在的 CPU 号，然后以此为下标从 tasklet_hi_vec[]中找到该 CPU 的队列头，把参数 t 所指向的 tasklet_struct 结构链入这个队列。由此可见，当某个 bh 函数被请求执行时，当前进程在哪个 CPU 上，就把这个 bh 函数“调度”到哪个 CPU 上执行。另一方面，tasklet_struct 代表着将要对 bh 函数的一次执行，在同一时间内，只能把它链入一个队列中，而不可能同时出现在多个队列中。对同一个 tasklet_struct 结构，如果已经对其调用了 tasklet_hi_schedule()函数，而尚未得到执行，就不允许再将其链入该队列，所以标志位 TASKLET_STATE_SCHED 就是保证这一点的。最后，通过 cpu_raise_softirq()发出软中断请求，其中的参数 HI_SOFTIRQ 表示 bh 与 HI_SOFTIRQ 软中断对应。

软中断 HI_SOFTIRQ 的服务例程为 tasklet_hi_action()：

```
static void tasklet_hi_action（struct softirq_action *a）
{
```

```
        int cpu = smp_processor_id();
        struct tasklet_struct *list;

        local_irq_disable();
        list = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = NULL;        /*临界区加锁*/
        local_irq_enable();

        while (list) {
                struct tasklet_struct *t = list;

                list = list->next;

                if (tasklet_trylock(t)) {
                        if (!atomic_read(&t->count)) {
                                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                                        BUG();
                                t->func(t->data);
                                tasklet_unlock(t);
                                continue;
                        }
                        tasklet_unlock(t);
                }

                local_irq_disable();
                t->next = tasklet_hi_vec[cpu].list;
                tasklet_hi_vec[cpu].list = t;
                __cpu_raise_softirq(cpu, HI_SOFTIRQ);
                local_irq_enable();
        }
}
```

这个函数除了加锁机制以外，读起来比较容易。其中要说明的是 t->func（t->data）语句，这条语句实际上就是调用 bh_action()函数：

```
/* BHs are serialized by spinlock global_bh_lock.

    It is still possible to make synchronize_bh() as
    spin_unlock_wait(&global_bh_lock). This operation is not used
   by kernel now, so that this lock is not made private only
    due to wait_on_irq().

    It can be removed only after auditing all the BHs.
  */
spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;

static void bh_action(unsigned long nr)
{
        int cpu = smp_processor_id();

        if (!spin_trylock(&global_bh_lock))
                goto resched;

        if (!hardirq_trylock(cpu))
```

```
                goto resched_unlock;

        if  (bh_base[nr])
                bh_base[nr]();

        hardirq_endlock (cpu) ;
        spin_unlock (&global_bh_lock) ;
       return;

resched_unlock:
        spin_unlock (&global_bh_lock) ;
resched:
        mark_bh (nr) ;
}
```

这里对 bh 函数的执行又设置了两道锁。一是 hardirq_trylock()，这是防止从一个硬中断内部调用 bh_action()。另一道锁是 spin_trylock()。这把锁就是全局量 global_bh_lock，只要有一个 CPU 在这个锁所锁住的临界区运行，别的 CPU 就不能进入这个区间，所以在任何时候最多只有一个 CPU 在执行 bh 函数。至于根据 bh 函数的编号执行相应的函数，那就比较容易理解了。

2. 软中断的执行

内核每当在 do_IRQ()中执行完一个中断请求队列中的中断服务例程以后，都要检查是否有软中断请求在等待执行。下面是 do_IRQ()中的一条语句：

```
if  (softirq_pending (cpu) )
             do_softirq();
```

在检测到软中断请求以后，就要通过 do_softirq()执行软中断服务例程，其代码在 /kernel/softirq.c 中：

```
asmlinkage void do_softirq()
{
        int cpu = smp_processor_id();
        __u32 pending;
        long flags;
        __u32 mask;

       if  (in_interrupt())
                return;

        local_irq_save (flags) ;/*把 eflags 寄存器的内容保存在 flags 变量中 * /

        pending = softirq_pending (cpu) ;

       if  (pending)  {
              struct softirq_action *h;

              mask = ~pending;
              local_bh_disable();
restart:
              /* Reset the pending bitmask before enabling irqs */
```

```
                softirq_pending（cpu） = 0;

                 local_irq_enable();  /*开中断*/

                 h = softirq_vec;

                 do {
                        if （pending & 1）
                                h->action（h）;
                         h++;
                         pending >>= 1;
                 } while （pending）;

                 local_irq_disable();  / *关中断*/

                pending = softirq_pending（cpu）;
                if （pending & mask） {
                        mask &= ~pending;
                         goto restart;
                 }
                 __local_bh_enable();

                if （pending)
                        wakeup_softirqd（cpu）;
        }

         local_irq_restore（flags）; / *恢复 eflags 寄存器的内容* /
}
```

从 do_softirq()的代码可以看出，使 CPU 不能执行软中断服务例程的“关卡”只有一个，那就是 in_interrupt()，这个宏限制了软中断服务例程既不能在一个硬中断服务例程内部执行，也不能在一个软中断服务例程内部执行（即嵌套）。但这个函数并没有对中断服务例程的执行进行“串行化”限制。这也就是说，不同的 CPU 可以同时进入对软中断服务例程的执行，每个 CPU 分别执行各自所请求的软中断服务。从这个意义上说，软中断服务例程的执行是“并发的”、“多序的”。但是，这些软中断服务例程的设计和实现必须十分小心，不能让它们相互干扰（例如通过共享的全局变量）。

从前面对软中断数据结构的介绍可以知道，尽管内核最多可以处理 32 个软中断，但目前只定义了 4 个软中断。在对软中断进行初始化时，soft_Init()函数只初始化了两个软中断 TASKLET_SOFTIRQ 和 HI_SOFTIRQ，这两个软中断对应的服务例程为 tasklet_action()和 tasklet_hi_action()。因此，do_softirq()中的 do_while 循环实际上是调用这两个函数。前面已经给出了 tasklet_hi_action()的源代码，而 tasklet_action()的代码与其基本一样，在此不再给出。

3.5.6　把 bh 移植到 tasklet

在 Linux 2.2 中，对中断的后半部分处理只提供了 bh 机制，而在 2.4 中新增加了两种机制：软中断和 tasklet。通过上面的介绍我们知道，同一个软中断服务例程可以同时在不

同的 CPU 上运行。为了提高 SMP 的性能，软中断现在主要用在网络子系统中。多个 tasklet 可以在多个不同的 CPU 上运行，但一个 CPU 一次只能处理一个 tasklet。bh 由内核进行了串行化处理，也就是在 SPM 环境中，某一时刻，一个 bh 函数只能由一个 CPU 来执行。如果要把 Linux 2.2 中的 bh 移植到 2.4 的 tasklet，请按下面方法进行。

1. Linux 2.4 中对 bh 的处理

假设一个 bh 为 FOO_BH（FOO 表示任意一个），其处理函数为 foo_bh，则：
（1）处理函数的原型为：void foo_bh（void）；
（2）通过 init_bh（FOO_BH, foo_bh）函数对 foo_bh 进行初始化；
（3）通过 mark_bh（FOO_BH）函数提出对 foo_bh()的执行请求。

2. 把 bh 移植到 tasklet

（1）处理函数的原型为：void foo_bh（unsigned long data）；
（2）通过宏 DECLARE_TASKLET_DISABLED（foo_tasklet, foo_bh, 0）或

```
struct tasklet_struct foo_tasklet;
tasklet_init（&foo_tasklet, foo_bh, 0）;
tasklet_disable（&foo_tasklet）;
```

对 foo_tasklet 进行初始化
（3）通过

```
tasklet_enable（&foo_tasklet）;
tasklet_schedule（&foo_tasklet）;
```

对 foo_tasklet 进行调度。

第四章　进程描述

操作系统中最核心的概念是进程，本章将对进程进行全面的描述。首先从程序的角度引入进程，接着对 Linux 中的进程进行了概要描述，在此基础上，对 Linux 中核心数据结构 task_struct 进行了较全面的介绍。另外，详细描述了内核对进程的 4 种组织方式，最后介绍了系统中一种特殊的进程 —— 内核线程。

4.1　进程和程序（Process and Program）

首先我们对进程作一明确定义：所谓进程是由正文段（Text）、用户数据段（User Segment）以及系统数据段（System Segment）共同组成的一个执行环境。

程序只是一个普通文件，是一个机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中，所以，程序是一个静态的实体。这里，对可执行映像做进一步解释，可执行映像就是一个可执行文件的内容，例如，你编写了一个 C 源程序，最终这个源程序要经过编译、连接成为一个可执行文件后才能运行。源程序中你要定义许多变量，在可执行文件中，这些变量就组成了数据段的一部分；源程序中的许多语句，例如“ i++; for(i=0; i<10; i++); ”等，在可执行文件中，它们对应着许多不同的机器代码指令，这些机器代码指令经 CPU 执行，就完成了你所期望的工作。可以这么说：程序代表你期望完成某工作的计划和步骤，它还浮在纸面上，等待具体实现。而具体的实现过程就是由进程来完成的，进程可以认为是运行中的程序，它除了包含程序中的所有内容外，还包含一些额外的数据。程序和进程的组成如图 4.1 所示。

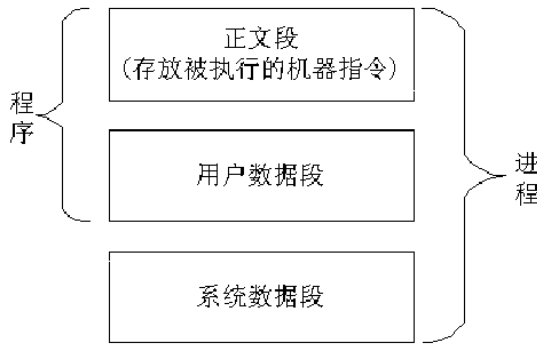


图 4.1 程序及进程的组成

程序装入内存后就可以运行了：在指令指针寄存器的控制下，不断地将指令取至 CPU 运

行。这些指令控制的对象不外乎各种存储器（内存、外存和各种 CPU 寄存器等），这些存储器中保存有待运行的指令和待处理的数据，当然，指令只有到 CPU 才能发挥其作用。可见，在计算机内部，程序的执行过程实际上就是一个执行环境的总和，这个执行环境包括程序中各种指令和数据，还有一些额外数据，比如说寄存器的值、用来保存临时数据（例如传递给某个函数的参数、函数的返回地址、保存变量等）的堆栈（包括程序堆栈和系统堆栈）、被打开文件的数量及输入输出设备的状态等。这个执行环境的动态变化表征程序的运行。我们就把这个环境称作“进程”，它代表程序的执行过程，是一个动态的实体，它随着程序中指令的执行而不断地变化。在某个特定时刻的进程的内容被称为进程映像（Process Image）。

Linux 是一个多任务操作系统，也就是说，可以有多个程序同时装入内存并运行，操作系统为每个程序建立一个运行环境即创建进程，每个进程拥有自己的虚拟地址空间，它们之间互不干扰，即使要相互作用（例如多个进程合作完成某个工作），也要通过内核提供的进程间通信机制（IPC）。Linux 内核支持多个进程虚拟地并发执行，这是通过不断地保存和切换程序的运行环境而实现的，选择哪个进程运行是由调度程序决定的。注意，在一些 UNIX 书籍中，又把“进程切换”（Process Switching）称为“环境切换”或“上下文切换”（Context Switching），这些术语表达的意思是相同的。

进程运行过程中，还需要其他的一些系统资源，例如，要用 CPU 来运行它的指令、要用系统的物理内存来容纳进程本身和它的有关数据、要在文件系统中打开和使用文件、并且可能直接或间接的使用系统的物理设备，例如打印机、扫描仪等。由于这些系统资源是由所有进程共享的，所以 Linux 必须监视进程和它所拥有的系统资源，使它们们可以公平地拥有系统资源以得到运行。

系统中最宝贵的资源是 CPU，通常来说，系统中只有一个 CPU，当然也可以有多个 CPU（Linux 支持 SMP___对称多处理机）。Linux 作为多任务操作系统，其目的就是让 CPU 上一直都有进程在运行，以最大限度地利用这一宝贵资源。通常情况下，进程数目是多于 CPU 数目的，这样其他进程必须等待 CPU 这一资源。操作系统通过调度程序来选择下一个最应该运行的进程，并使用一系列的调度策略来确保公平和高效。

进程是一个动态实体，如图 4.1 所示，进程实体由 3 个独立的部分组成。

（1）正文段（Text）：存放被执行的机器指令。这个段是只读的（所以，在这里不能写自己能修改的代码），它允许系统中正在运行的两个或多个进程之间能够共享这一代码。例如，有几个用户都在使用文本编辑器，在内存中仅需要该程序指令的一个副本，他们全都共享这一副本。

（2）用户数据段（User Segment）：存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里包含的信息可以被改变。虽然进程之间可以共享正文段，但是每个进程需要有它自己的专用用户数据段。例如同时编辑文本的用户，虽然运行着同样的程序编辑器，但是每个用户都有不同的数据：正在编辑的文本。

（3）系统数据段（System Segment）：该段有效地存放程序运行的环境。事实上，这正是程序和进程的区别所在。如前所述，程序是由一组指令和数据组成的静态事物，它们是进程最初使用的正文段和用户数据段。作为动态事物，进程是正文段、用户数据段和系统数据段的信息的交叉综合体，其中系统数据段是进程实体最重要的一部分，之所以说它有效地存放程序运行的环境，是因为这一部分存放有进程的控制信息。系统中有许多进程，操作系统

要管理它们、调度它们运行，就是通过这些控制信息。Linux 为每个进程建立了 task_struct 数据结构来容纳这些控制信息。

总之，进程是一个程序完整的执行环境。该环境是由正文段、用户数据段、系统数据段的信息交织在一起组成的。

4.2　Linux 中的进程概述

Linux 中的每个进程由一个 task_struct 数据结构来描述，在 Linux 中，任务（Task）和进程（Process）是两个相同的术语，task_struct 其实就是通常所说的“进程控制块”即 PCB。task_struct 容纳了一个进程的所有信息，是系统对进程进行控制的唯一手段，也是最有效的手段。

在 Linux 2.4 中，Linux 为每个新创建的进程动态地分配一个 task_struct 结构。系统所允许的最大进程数是由机器所拥有的物理内存的大小决定的,例如,在 IA32 的体系结构中，一个 512MB 内存的机器，其最大进程数可以达到 32KB，这是对旧内核（2.2 以前）版本的极大改进[①]。

Linux 支持多处理机（SMP），所以系统中允许有多个 CPU。Linux 作为多处理机操作系统时，系统中允许的最大 CPU 个数为 32。很显然，Linux 作为单机操作系统时，系统中只有一个 CPU，本书主要讨论单处理机的情况。

和其他操作系统类似，Linux 也支持两种进程：普通进程和实时进程。实时进程具有一定程度上的紧迫性，要求对外部事件做出非常快的响应；而普通进程则没有这种限制。所以，调度程序要区分对待这两种进程，通常，实时进程要比普通进程优先运行。这两种进程的区分也反映在 task_struct 数据结构中了。

总之，包含进程所有信息的 task_struct 数据结构是比较庞大的，但是该数据结构本身并不复杂，我们将它的所有域按其功能可做如下划分：

- 进程状态（State）；
- 进程调度信息（Scheduling Information）；
- 各种标识符（Identifiers）；
- 进程通信有关信息（IPC，Inter_Process Communication）；
- 时间和定时器信息（Times and Timers）；
- 进程链接信息（Links）；
- 文件系统信息（File System）；
- 虚拟内存信息（Virtual Memory）；
- 页面管理信息（page）；
- 对称多处理器（SMP）信息；
- 和处理器相关的环境（上下文）信息（Processor Specific Context）；

[①] 在 Linux 2.2 及以前的版本中，用一个 task 数组来管理系统中所有进程的 task_struct 结构，因此，系统中进程的最大个数受数组大小的限制。

- 其他信息。

下面我们对 task_struct 结构进行具体描述。

4.3 task_struct 结构描述

1. 进程状态（State）

进程执行时，它会根据具体情况改变状态。进程状态是调度和对换的依据。Linux 中的进程主要有如下状态，如表 4.1 所示。

表 4.1　　Linux 进程的状态

内核表示	含义
TASK_RUNNING	可运行
TASK_INTERRUPTIBLE	可中断的等待状态
TASK_UNINTERRUPTIBLE	不可中断的等待状态
TASK_ZOMBIE	僵死
TASK_STOPPED	暂停
TASK_SWAPPING	换入/换出

（1）可运行状态

处于这种状态的进程，要么正在运行、要么正准备运行。正在运行的进程就是当前进程（由 current 所指向的进程），而准备运行的进程只要得到 CPU 就可以立即投入运行，CPU 是这些进程唯一等待的系统资源。系统中有一个运行队列（run_queue），用来容纳所有处于可运行状态的进程，调度程序执行时，从中选择一个进程投入运行。在后面我们讨论进程调度的时候，可以看到运行队列的作用。当前运行进程一直处于该队列中，也就是说，current 总是指向运行队列中的某个元素，只是具体指向谁由调度程序决定。

（2）等待状态

处于该状态的进程正在等待某个事件（Event）或某个资源，它肯定位于系统中的某个等待队列（wait_queue）中。Linux 中处于等待状态的进程分为两种：可中断的等待状态和不可中断的等待状态。处于可中断等待态的进程可以被信号唤醒，如果收到信号，该进程就从等待状态进入可运行状态，并且加入到运行队列中，等待被调度；而处于不可中断等待态的进程是因为硬件环境不能满足而等待，例如等待特定的系统资源，它任何情况下都不能被打断，只能用特定的方式来唤醒它，例如唤醒函数 wake_up（）等。

（3）暂停状态

此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 信号后就处于这种状态。例如，正接受调试的进程就处于这种状态。

（4）僵死状态

进程虽然已经终止，但由于某种原因，父进程还没有执行 wait()系统调用，终止进程的信息也还没有回收。顾名思义，处于该状态的进程就是死进程，这种进程实际上是系统中的垃圾，必须进行相应处理以释放其占用的资源。

2．进程调度信息

调度程序利用这部分信息决定系统中哪个进程最应该运行，并结合进程的状态信息保证系统运转的公平和高效。这一部分信息通常包括进程的类别（普通进程还是实时进程）、进程的优先级等，如表 4.2 所示。

表 4.2　进程调度信息

域名	含义
need_resched	调度标志
Nice	静态优先级
Counter	动态优先级
Policy	调度策略
rt_priority	实时优先级

在下一章的进程调度中我们会看到，当 need_resched 被设置时，在“下一次的调度机会”就调用调度程序 schedule()counter 代表进程剩余的时间片，是进程调度的主要依据，也可以说是进程的动态优先级，因为这个值在不断地减少；nice 是进程的静态优先级，同时也代表进程的时间片，用于对 counter 赋值，可以用 nice()系统调用改变这个值；policy 是适用于该进程的调度策略，实时进程和普通进程的调度策略是不同的；rt_priority 只对实时进程有意义，它是实时进程调度的依据。

进程的调度策略有 3 种，如表 4.3 所示。

表 4.3　进程调度的策略

名称	解释	适用范围
SCHED_OTHER	其他调度	普通进程
SCHED_FIFO	先来先服务调度	实时进程
SCHED_RR	时间片轮转调度	

只有 root 用户能通过 sched_setscheduler（）系统调用来改变调度策略。

3．标识符（Identifiers）

每个进程有进程标识符、用户标识符、组标识符，如表 4.4 所示。

不管对内核还是普通用户来说，怎么用一种简单的方式识别不同的进程呢？这就引入了进程标识符（PID，process identifier），每个进程都有一个唯一的标识符，内核通过这个

标识符来识别不同的进程，同时，进程标识符 PID 也是内核提供给用户程序的接口，用户程序通过 PID 对进程发号施令。PID 是 32 位的无符号整数，它被顺序编号：新创建进程的 PID 通常是前一个进程的 PID 加 1。然而，为了与 16 位硬件平台的传统 Linux 系统保持兼容，在 Linux 上允许的最大 PID 号是 32767，当内核在系统中创建第 32768 个进程时，就必须重新开始使用已闲置的 PID 号。

表 4.4　　　　各种标识符

域名	含义
Pid	进程标识符
Uid、gid	用户标识符、组标识符
Euid、egid	有效用户标识符、有效组标识符
Suid、sgid	备份用户标识符、备份组标识符
Fsuid、fsgid	文件系统用户标识符、文件系统组标识符

另外，每个进程都属于某个用户组。task_struct 结构中定义有用户标识符和组标识符。它们同样是简单的数字，这两种标识符用于系统的安全控制。系统通过这两种标识符控制进程对系统中文件和设备的访问，其他几个标识符将在文件系统中讨论。

4．进程通信有关信息（IPC，Inter_Process Communication）

为了使进程能在同一项任务上协调工作，进程之间必须能进行通信即交流数据。

Linux 支持多种不同形式的通信机制。它支持典型的 UNIX 通信机制（IPC Mechanisms）：信号（Signals）、管道（Pipes），也支持 System V 通信机制：共享内存（Shared Memory）、信号量和消息队列（Message Queues）,如表 4.5 所示。

表 4.5　　　　进程通信有关信息

域名	含义
Spinlock_t sigmask_lock	信号掩码的自旋锁
Long blocked	信号掩码
Struct signal　*sig	信号处理函数
Struct sem_undo *semundo	为避免死锁而在信号量上设置的取消操作
Struct sem_queue *semsleeping	与信号量操作相关的等待队列

这些域的具体含义将在进程通信一章进行讨论。

5．进程链接信息（Links）

程序创建的进程具有父/子关系。因为一个进程能创建几个子进程，而子进程之间有兄弟关系，在 task_struct 结构中有几个域来表示这种关系。

在 Linux 系统中，除了初始化进程 init，其他进程都有一个父进程（Parent Process）或称为双亲进程。可以通过 fork()或 clone()系统调用来创建子进程，除了进程标识符（PID）等必要的信息外，子进程的 task_struct 结构中的绝大部分的信息都是从父进程中拷贝，或说“克隆”过来的。系统有必要记录这种“亲属”关系，使进程之间的协作更加方便，例如父进程给子进程发送杀死（kill）信号、父子进程通信等，就可以用这种关系很方便地实现。

每个进程的 task_struct 结构有许多指针，通过这些指针，系统中所有进程的 task_struct 结构就构成了一棵进程树，这棵进程树的根就是初始化进程 init 的 task_struct 结构（init 进程是 Linux 内核建立起来后人为创建的一个进程，是所有进程的祖先进程）。表 4.6 是进程所有的链接信息。

表 4.6 进程链接信息

名称	英文解释	中文解释［指向哪个进程］
p_opptr	Original parent	祖先
p_pptr	Parent	父进程
p_cptr	Child	子进程
p_ysptr	Younger sibling	弟进程
p_osptr	Older sibling	兄进程
Pidhash_next、 Pidhash_pprev		进程在哈希表中的链接
Next_task、 prev_task		进程在双向循环链表中的链接
Run_list		运行队列的链表

6．时间和定时器信息（Times and Timers）

一个进程从创建到终止叫做该进程的生存期（lifetime）。进程在其生存期内使用 CPU 的时间，内核都要进行记录，以便进行统计、计费等有关操作。进程耗费 CPU 的时间由两部分组成：一是在用户模式（或称为用户态）下耗费的时间、一是在系统模式（或称为系统态）下耗费的时间。每个时钟滴答，也就是每个时钟中断，内核都要更新当前进程耗费 CPU 的时间信息。

“时间”对操作系统是极其重要的 。读者可能了解计算机时间的有关知识，例如 8353/8254 这些物理器件，INT 08、INT 1C 等时钟中断等，可能有过编程序时截获时钟中断的成就感，不管怎样，下一章我们将用较大的篇幅尽可能向读者解释清楚操作系统怎样建立完整的时间机制、并在这种机制的激励下进行调度等活动。

建立了“时间”的概念，“定时”就是轻而易举的了，无非是判断系统时间是否到达某个时刻，然后执行相关的操作而已。Linux 提供了许多种定时方式，用户可以灵活使用这些方式来为自己的程序定时。

表 4.7 是和时间有关的域，上面所说的 counter 是指进程剩余的 CPU 时间片，也和时间

有关，所以这里我们再次提及它。表 4.8 是进程的所有定时器。

表 4.7　　　　与时间有关的域

域名	含义
Start_time	进程创建时间
Per_cpu_utime	进程在某个 CPU 上运行时在用户态下耗费的时间
Per_cpu_stime	进程在某个 CPU 上运行时在系统态下耗费的时间
Counter	进程剩余的时间片

表 4.8　　　　进程的所有定时器

<table>
<tr><th>定时器类型</th><th>解释</th><th>什么时候更新</th><th>用来表示此种定时器的域</th></tr>
<tr><td rowspan="3">ITIMER_REAL</td><td rowspan="3">实时定时器</td><td rowspan="3">实时更新，即不论该进程是否运行</td><td>it_real_value</td></tr>
<tr><td>it_real_incr</td></tr>
<tr><td>real_timer</td></tr>
<tr><td rowspan="2">ITIMER_VIRTUAL</td><td rowspan="2">虚拟定时器</td><td rowspan="2">只在进程运行于用户态时更新</td><td>it_virt_value</td></tr>
<tr><td>it_virt_incr</td></tr>
<tr><td rowspan="2">ITIMER_PROF</td><td rowspan="2">概况定时器</td><td rowspan="2">进程运行于用户态和系统态时更新</td><td>it_prof_value</td></tr>
<tr><td>it_prof_incr</td></tr>
</table>

进程有 3 种类型的定时器：实时定时器、虚拟定时器和概况定时器。这 3 种定时器的特征共有 3 个：到期时间、定时间隔和要触发的事件。到期时间就是定时器到什么时候完成定时操作，从而触发相应的事件；定时间隔就是两次定时操作的时间间隔，它决定了定时操作是否继续进行，如果定时间隔大于 0，则在定时器到期时，该定时器的到期时间被重新赋值，使定时操作继续进行下去，直到进程结束或停止使用定时器，只不过对不同的定时器，到期时间的重新赋值操作是不同的。在表 4.8 中，每个定时器都有两个域来表示到期时间和定时间隔：value 和 incr，二者的单位都是时钟滴答，和 jiffies 的单位是一致的，Linux 所有的时间应用都建立在 jiffies 之上。虚拟定时器和概况定时器到期时由内核发送相应的信号，而实时定时器比较特殊，它由内核机制提供支持，我们将在后面讨论这个问题。

每个时钟中断，当前进程所有和时间有关的信息都要更新：当前进程耗费的 CPU 时间要更新，以便于最后的计费；时间片计数器 counter 要更新，如果 counter<=0，则要执行调度程序；进程申请的延时要更新，如果延时时间到了，则唤醒该进程；所有的定时器都要更新，Linux 内核检测这些定时器是否到期，如果到期，则执行相应的操作。在这里，“更新”的具体操作是不同的：对 counter，内核要对它减值，而对于所有的定时器，就是检测它的值，内核把系统当前时间和其到期时间作一比较，如果到期时间小于系统时间，则表示该定时器到期。但为了方便，我们把这些操作一概称为“更新”，请读者注意。

请特别注意上面 3 个定时器的更新时间。实时定时器不管其所属的进程是否运行都要更新，所以，时钟中断来临时，系统中所有进程的实时定时器都被更新，如果有多个进程的实

时定时器到期，则内核要一一处理这些定时器所触发的事件。而虚拟定时器和概况定时器只在进程运行时更新，所以，时钟中断来临时，只有当前进程的概况定时器得到更新，如果当前进程运行于用户态，则其虚拟定时器也得到更新。

此外，Linux 内核对这 3 种定时器的处理是不同的，虚拟定时器和概况定时器到期时，内核向当前进程发送相应的信号：SIGVTALRM 、SIGPROF ；而实时定时器要执行的操作由 real_timer 决定，real_time 是 timer_list 类型的变量（定义：struct timer_list real_timer），其中容纳了实时定时器的到期时间、定时间隔等信息，我们将在下一章详细讨论这些内容。

7．文件系统信息（File System）

进程可以打开或关闭文件，文件属于系统资源，Linux 内核要对进程使用文件的情况进行记录。task_struct 结构中有两个数据结构用于描述进程与文件相关的信息。其中，fs_struct 中描述了两个 VFS 索引节点（VFS inode），这两个索引节点叫做 root 和 pwd，分别指向进程的可执行映像所对应的根目录（Home Directory）和当前目录或工作目录。file_struct 结构用来记录了进程打开的文件的描述符（Descriptor）。如表 4.9 所示。

表 4.9　与文件系统相关的域

定义形式	解释
Sruct fs_struct *fs	进程的可执行映像所在的文件系统
Struct files_struct *files	进程打开的文件

在文件系统中，每个 VFS 索引节点唯一描述一个文件或目录，同时该节点也是向更低层的文件系统提供的统一的接口。

8．虚拟内存信息（Virtual Memory）

除了内核线程（Kernel Thread），每个进程都拥有自己的地址空间（也叫虚拟空间），用 mm_struct 来描述。另外 Linux 2.4 还引入了另外一个域 active_mm，这是为内核线程而引入的。因为内核线程没有自己的地址空间，为了让内核线程与普通进程具有统一的上下文切换方式，当内核线程进行上下文切换时，让切换进来的线程的 active_mm 指向刚被调度出去的进程的 active_mm（如果进程的 mm 域不为空，则其 active_mm 域与 mm 域相同）。内存信息如表 4.10 所示。

表 4.10　虚拟内存描述信息

定义形式	解释
Struct mm_struct *mm	描述进程的地址空间
Struct mm_struct *active_mm	内核线程所借用的地址空间

9．页面管理信息

当物理内存不足时，Linux 内存管理子系统需要把内存中的部分页面交换到外存，其交换是以页为单位的。有关页面的描述信息如表 4.11。

表 4.11　　页面管理信息

定义形式	解释
Int swappable	进程占用的内存页面是否可换出
Unsigned long min_flat,maj_flt,nswap	进程累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数
Unsigned long cmin_flat,cmaj_flt,cnswap	本进程作为祖先进程，其所有层次子进程的累计的次（minor）缺页次数、主(major)次数及累计换出、换入页面数

10．对称多处理机（SMP）信息

Linux 2.4 对 SMP 进行了全面的支持，表 4.12 是与多处理机相关的几个域。

表 4.12　　与多处理机相关的域

定义形式	解释
Int has_cpu	进程当前是否拥有 CPU
Int processor	进程当前正在使用的 CPU
Int lock_depth	上下文切换时内核锁的深度

11．和处理器相关的环境（上下文）信息（Processor Specific Context）

这里要特别注意标题：和“处理器”相关的环境信息。进程作为一个执行环境的综合，当系统调度某个进程执行，即为该进程建立完整的环境时，处理器（Processor）的寄存器、堆栈等是必不可少的。因为不同的处理器对内部寄存器和堆栈的定义不尽相同，所以叫做“和处理器相关的环境”，也叫做“处理机状态”。当进程暂时停止运行时，处理机状态必须保存在进程的 task_struct 结构中，当进程被调度重新运行时再从中恢复这些环境，也就是恢复这些寄存器和堆栈的值。处理机信息如表 4.13 所示。

表 4.13　　与处理机相关的信息

定义形式	解释
Struct thread_struct *tss	任务切换状态

12．其他

（1）struct wait_queue *wait_chldexit

在进程结束时,或发出系统调用 wait4 时，为了等待子进程的结束，而将自己（父进程）睡眠在该等待队列上，设置状态标志为 TASK_INTERRUPTIBLE，并且把控制权转给调度程序。

（2）Struct rlimit rlim[RLIM_NLIMITS]

每一个进程可以通过系统调用 setlimit 和 getlimit 来限制它资源的使用。

（3）Int exit_code exit_signal

程序的返回代码以及程序异常终止产生的信号，这些数据由父进程（子进程完成后）轮流查询。

（4）Char comm[16]

这个域存储进程执行的程序的名字，这个名字用在调试中。

（5）Unsigned long personality

Linux 可以运行 X86 平台上其他 UNIX 操作系统生成的符合 iBCS2 标准的程序，personality 进一步描述进程执行的程序属于何种 UNIX 平台的“个性”信息。通常有 PER_Linux,PER_Linux_32BIT,PER_Linux_EM86,PER_SVR4,PER_SVR3,PER_SCOSVR3,PER_WYSEV386,PER_ISCR4,PER_BSD,PER_XENIX 和 PER_MASK 等,参见 include／Linux/personality.h>。

（6）int did_exec:1

按 POSIX 要求设计的布尔量，区分进程正在执行老程序代码，还是用系统调用 execve（）装入一个新的程序。

（7）struct linux_binfmt *binfmt

指向进程所属的全局执行文件格式结构，共有 a.out、script、elf、java 等 4 种。

综上所述，我们对进程的 task_struct 结构进行了归类讨论，还有一些域没有涉及到，在第六章进程的创建与执行一节几乎涉及到所有的域，在那里可以对很多域有更深入一步的理解。task_struct 结构是进程实体的核心，Linux 内核通过该结构来控制进程：首先通过其中的调度信息决定该进程是否运行；当该进程运行时，根据其中保存的处理机状态信息来恢复进程运行现场，然后根据虚拟内存信息，找到程序的正文和数据；通过其中的通信信息和其他进程实现同步、通信等合作。几乎所有的操作都要依赖该结构，所以，task_struct 结构是一个进程存在的唯一标志。

4.4　task_struct 结构在内存中的存放

task_struct 结构在内存的存放与内核栈是分不开的，因此，首先讨论内核栈。

4.4.1　进程内核栈

每个进程都有自己的内核栈。当进程从用户态进入内核态时，CPU 就自动地设置该进程的内核栈，也就是说，CPU 从任务状态段 TSS 中装入内核栈指针 esp（参见下一章的进程切换一节）。

X86 内核栈的分布如图 4.2 所示。

在 Intel 系统中，栈起始于末端，并朝这个内存区开始的方向增长。从用户态刚切换到内核态以后，进程的内核栈总是空的，因此，esp 寄存器直接指向这个内存区的顶端在图 4.2 中，从用户态切换到内核态后，esp 寄存器包含的地址为 0x018fc00。进程描述符存放在从

0x015fa00 开始的地址。只要把数据写进栈中，esp 的值就递减。

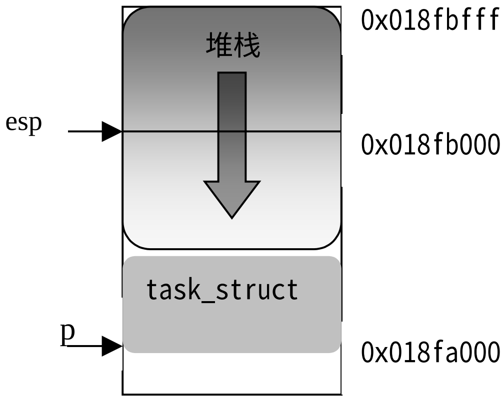


图 4.2　内核栈的分布图

在／include/linux/sched.h 中定义了如下一个联合结构：

```
    union task_union {
       struct task_struct task;
       unsigned long stack[2408];
};
```

从这个结构可以看出，内核栈占 8KB 的内存区。实际上，进程的 task_struct 结构所占的内存是由内核动态分配的，更确切地说，内核根本不给 task_struct 分配内存，而仅仅给内核栈分配 8KB 的内存，并把其中的一部分给 task_struct 使用。

task_struct 结构大约占 1K 字节左右，其具体数字与内核版本有关，因为不同的版本其域稍有不同。因此，内核栈的大小不能超过 7KB，否则，内核栈会覆盖 task_struct 结构，从而导致内核崩溃。不过，7KB 大小对内核栈已足够。

把 task_struct 结构与内核栈放在一起具有以下好处：

- 内核可以方便而快速地找到这个结构，用伪代码描述如下：

task_struct = (struct task_struct *) STACK_POINTER & 0xffffe000

- 避免在创建进程时动态分配额外的内存。
- task_struct 结构的起始地址总是开始于页大小（PAGE_SIZE）的边界。

4.4.2　当前进程（current 宏）

当一个进程在某个 CPU 上正在执行时，内核如何获得指向它的 task_struct 的指针？上面所提到的存储方式为达到这一目的提供了方便。在 linux/include/ i386／current.h 中定义了 current 宏，这是一段与体系结构相关的代码：

```
tatic inline struct task_struct * get_current (void)
{
struct task_struct *current;
__asm__ ("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL) ) ;
return current;
}
```

实际上，这段代码相当于如下一组汇编指令（设 p 是指向当前进程 task_struc 结构的指针）：

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

换句话说，仅仅只需检查栈指针的值，而根本无需存取内存，内核就可以导出 task_struct 结构的地址。

在本书的描述中，会经常出现 current 宏，在内核代码中也随处可见，可以把它看作全局变量来用，例如，current->pid 返回在 CPU 上正在执行的进程的标识符。

另外，在 include/ i386/processor.h 中定义了两个函数 free_task_struct(　)和 alloc_task_struct(　)，前一个函数释放 8KB 的 task_union 内存区，而后一个函数分配 8KB 的 task_union 内存区。

4.5　进程组织方式

在 Linux 中，可以把进程分为用户任务和内核线程，不管是哪一种进程，它们都有自己的 task_struct。在 2.4 版中，系统拥有的进程数可能达到数千乃至上万个，尤其对于企业级应用（如数据库应用及网络服务器）更是如此。为了对系统中的很多进程及处于不同状态的进程进行管理，Linux 采用了如下几种组织方式。

4.5.1　哈希表

哈希表是进行快速查找的一种有效的组织方式。Linux 在进程中引入的哈希表叫做 pidhash，在 include/linux/sched.h 中定义如下：

```
#define PIDHASH_SZ  (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];

#define pid_hashfn (x)     ( ( ( (x)  >> 8)  ^  (x) )  &  (PIDHASH_SZ - 1) )
```

其中，PIDHASH_SZ 为表中元素的个数，表中的元素是指向 task_struct 结构的指针。pid_hashfn 为哈希函数，把进程的 PID 转换为表的索引。通过这个函数，可以把进程的 PID 均匀地散列在它们的域（0 到 PID_MAX-1）中。

在数据结构课程中我们已经了解到，哈希函数并不总能确保 PID 与表的索引一一对应，两个不同的 PID 散列到相同的索引称为冲突。

Linux 利用链地址法来处理冲突的 PID：也就是说，每一表项是由冲突的 PID 组成的双向链表，这种链表是由 task_struct 结构中的 pidhash_next 和 pidhash_pprev 域实现的，同一链表中 pid 的大小由小到大排列。如图 4.3 所示。

哈希表 pidhash 中插入和删除一个进程时可以调用 hash_ pid(　) 和 unhash_ pid(　) 函数。对于一个给定的 pid，可以通过 find_task_by_pid()函数快速地找到对应的进程：

```
static inline struct task_struct *find_task_by_pid (int pid)
{
```

```
        struct task_struct *p, **htable = &pidhash[pid_hashfn (pid) ];
        for (p = *htable; p && p->pid != pid; p = p->pidhash_next)
                ;
        return p;
}
```

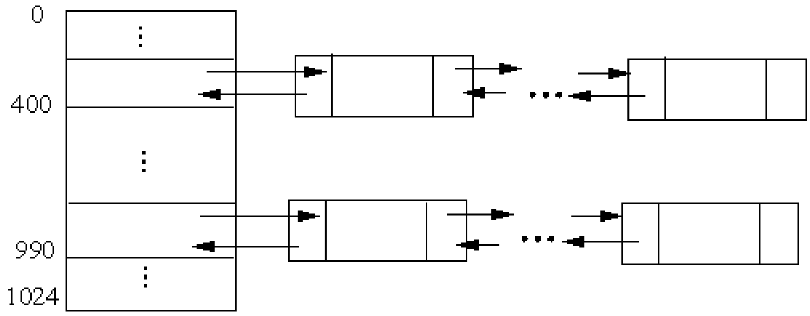


图 4.3　链地址法处理冲突时的哈希表

4.5.2　双向循环链表

哈希表的主要作用是根据进程的 pid 可以快速地找到对应的进程，但它没有反映进程创建的顺序，也无法反映进程之间的亲属关系，因此引入双向循环链表。每个进程 task_struct 结构中的 prev_task 和 next_task 域用来实现这种链表，如图 4.4 所示。

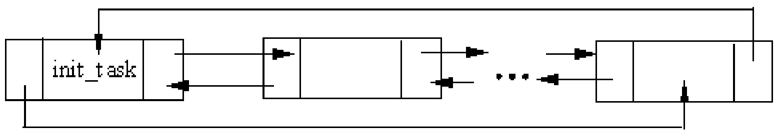


图 4.4　双向循环链表

宏 SET_LINK 用来在该链表中插入一个元素：

```
#define SET_LINKS (p)  do { \
       (p) ->next_task = &init_task; \
       (p) ->prev_task = init_task.prev_task; \
      init_task.prev_task->next_task =  (p) ; \
      init_task.prev_task =  (p) ; \
       (p) ->p_ysptr = NULL; \
       if  ( ( (p) ->p_osptr =  (p) ->p_pptr->p_cptr)  != NULL)  \
               (p) ->p_osptr->p_ysptr = p; \
        (p) ->p_pptr->p_cptr = p; \
       } while  (0)
```

从这段代码可以看出，链表的头和尾都为 init_task，它对应的是进程 0（pid 为 0），也就是所谓的空进程，它是所有进程的祖先。这个宏把进程之间的亲属关系也链接起来。另外，还有一个宏 for_each_task()：

```
#define for_each_task (p)  \
        for  (p = &init_task ;  (p = p->next_task)  != &init_task ; )
```

这个宏是循环控制语句。注意 init_task 的作用，因为空进程是一个永远不存在的进程，因此用它做链表的头和尾是安全的。

因为进程的双向循环链表是一个临界资源，因此在使用这个宏时一定要加锁，使用完后开锁。

4.5.3　运行队列

当内核要寻找一个新的进程在 CPU 上运行时，必须只考虑处于可运行状态的进程（即在 TASK_RUNNING 状态的进程），因为扫描整个进程链表是相当低效的，所以引入了可运行状态进程的双向循环链表，也叫运行队列（runqueue）。

运行队列容纳了系统中所有可以运行的进程，它是一个双向循环队列，其结构如图 4.5 所示。

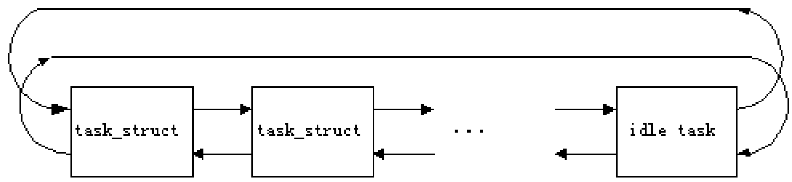


图 4.5　运行队列的结构

4.5.4　进程的运行队列链表

该队列通过 task_struct 结构中的两个指针 run_list 链表来维持。队列的标志有两个：一个是“空进程”idle_task，一个是队列的长度。

有两个特殊的进程永远在运行队列中待着：当前进程和空进程。前面我们讨论过，当前进程就是由 cureent 指针所指向的进程，也就是当前运行着的进程，但是请注意，current 指针在调度过程中（调度程序执行时）是没有意义的，为什么这么说呢？调度前，当前进程正在运行，当出现某种调度时机引发了进程调度，先前运行着的进程处于什么状态是不可知的，多数情况下处于等待状态，所以这时候 current 是没有意义的，直到调度程序选定某个进程投入运行后，current 才真正指向了当前运行进程；空进程是个比较特殊的进程，只有系统中没有进程可运行时它才会被执行，Linux 将它看作运行队列的头，当调度程序遍历运行队列，是从 idle_task 开始、至 idle_task 结束的，在调度程序运行过程中，允许队列中加入新出现的可运行进程，新出现的可运行进程插入到队尾，这样的好处是不会影响到调度程序所要遍历的队列成员，可见，idle_task 是运行队列很重要的标志。

另一个重要标志是队列长度，也就是系统中处于可运行状态（TASK_RUNNING）的进程数目，用全局整型变量 nr_running 表示，在/kernel/fork.c 中定义如下：

```
int nr_running=1;
```

若 nr_running 为 0，就表示队列中只有空进程。在这里要说明一下：若 nr_running 为 0，则系统中的当前进程和空进程就是同一个进程。但是 Linux 会充分利用 CPU 而尽量避免出

现这种情况。

4.5.5　等待队列

在 2.4 版本中，引入了一种特殊的链表——通用双向链表，它是内核中实现其他链表的基础，也是面向对象的思想在 C 语言中的应用。在等待队列的实现中多次涉及与此链表相关的内容。

1．通用双向链表

在 include/linux/list.h 中定义了这种链表：

```
struct list_head {
        struct list_head *next, *prev;
};
```

这是双向链表的一个基本框架，在其他使用链表的地方就可以使用它来定义任意一个双向链表，例如：

```
struct foo_list {
     int data;
     struct list_head list;
};
```

对于 list_head 类型的链表，Linux 定义了 5 个宏：

```
#define LIST_HEAD_INIT (name)  { & (name) , & (name)  }

#define LIST_HEAD (name)  \
        struct list_head name = LIST_HEAD_INIT (name)

#define INIT_LIST_HEAD (ptr)  do { \
         (ptr) ->next =  (ptr) ;  (ptr) ->prev =  (ptr) ; \
} while  (0)

#define list_entry (ptr, type, member)  \
         ( (type *)  ( (char *)  (ptr) - (unsigned long)  (& ( (type *) 0) ->member) ) )

#define list_for_each (pos, head)  \
        for  (pos =  (head) ->next; pos !=  (head) ; pos = pos->next)
```

前 3 个宏都是初始化一个空的链表，但用法不同，LIST_HEAD_INIT()在声明时使用，用来初始化结构元素，第 2 个宏用在静态变量初始化的声明中，而第 3 个宏用在函数内部。

其中，最难理解的宏为 list_entry（），在内核代码的很多处都用到这个宏，例如，在调度程序中，从运行队列中选择一个最值得运行的进程，部分代码如下：

```
static LIST_HEAD (runqueue_head) ;
struct list_head *tmp;
struct task_struct *p;

list_for_each (tmp, &runqueue_head)  {
    p = list_entry (tmp, struct task_struct, run_list) ;
```

```
        if (can_schedule (p) ) {
            int weight = goodness (p, this_cpu, prev->active_mm) ;
            if (weight > c)
                c = weight, next = p;
        }
    }
```

从这段代码可以分析出 list_entry(ptr，type，member)宏及参数的含义：　ptr　是指向 list_ head 类型链表的指针，type 为一个结构，而 member 为结构 type 中的一个域，类型为 list_head，这个宏返回指向 type 结构的指针。在内核代码中大量引用了这个宏，因此，搞清楚这个宏的含义和用法非常重要。

另外，对 list_head 类型的链表进行删除和插入（头或尾）的宏为 list_del()/list_add()/list_ add_tail()，在内核的其他函数中可以调用这些宏。例如，从运行队列中删除、增加及移动一个任务的代码如下：

```
static inline void del_from_runqueue (struct task_struct * p)
{
        nr_running--;
        list_del (&p->run_list) ;
        p->run_list.next = NULL;
}
static inline void add_to_runqueue (struct task_struct * p)
{
        list_add (&p->run_list, &runqueue_head) ;
        nr_running++;
}

static inline void move_last_runqueue (struct task_struct * p)
{
        list_del (&p->run_list) ;
        list_add_tail (&p->run_list, &runqueue_head) ;
}

static inline void move_first_runqueue (struct task_struct * p)
{
        list_del (&p->run_list) ;
        list_add (&p->run_list, &runqueue_head) ;
}
```

2．等待队列

运行队列链表把处于 TASK_RUNNING 状态的所有进程组织在一起。当要把其他状态的进程分组时，不同的状态要求不同的处理，Linux 选择了下列方式之一。

- TASK_STOPPED 或　TASK_ZOMBIE 状态的进程不链接在专门的链表中,也没必要把它们分组，因为父进程可以通过进程的 PID 或进程间的亲属关系检索到子进程。
- 把 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态的进程再分成很多类，其每一类对应一个特定的事件。在这种情况下，进程状态提供的信息满足不了快速检索进程，因此，有必要引入另外的进程链表。这些链表叫等待队列。

等待队列在内核中有很多用途，尤其对中断处理、进程同步及定时用处更大。因为这些

内容在以后的章节中讨论，我们只在这里说明，进程必须经常等待某些事件的发生，例如，等待一个磁盘操作的终止，等待释放系统资源或等待时间走过固定的间隔。等待队列实现在事件上的条件等待，也就是说，希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。等待队列由循环链表实现。在 2.4 版中，关于等待队列的定义如下（为了描述方便，有所简化）：

```
struct __wait_queue {
        unsigned int flags;
        struct task_struct * task;
        struct list_head task_list;
} ;
typedef struct __wait_queue wait_queue_t ;
```

另外，关于等待队列另一个重要的数据结构——等待队列首部的描述如下：

```
struct __wait_queue_head {
         wq_lock_t lock;
        struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

在这两个数据结构的定义中，都涉及到类型为 list_head 的链表，这与 2.2 版定义是不同的，在 2.2 版中的定义为：

```
struct wait_queue {
        struct task_struct * task;
         struct wait_queue * next ;
} ;
typedef struct wait_queue wait_queue_t ;
typedef struct wait_queue *wait_queue_head_t ;
```

这里要特别强调的是，2.4 版中对等待队列的操作函数和宏比 2.2 版丰富了，而在你编写设备驱动程序时会用到这些函数和宏，因此，要注意 2.2 到 2.4 函数的移植问题。下面给出 2.4 版中的一些主要函数及其功能：

- init_waitqueue_head（）——对等待队列首部进行初始化
- init_waitqueue_entry（）——对要加入等待队列的元素进行初始化
- waitqueue_active（）——判断等待队列中已经没有等待的进程
- add_wait_queue（）——给等待队列中增加一个元素
- remove_wait_queue（）——从等待队列中删除一个元素

注意，在以上函数的实现中，都调用了对 list_head 类型链表的操作函数（list_del()/list_add()/list_add_tail()），因此可以说，list_head 类型相当于 C++中的基类型，这也是对 2.2 版的极大改进。

希望等待一个特定事件的进程能调用下列函数中的任一个：

sleep_on( )函数对当前的进程起作用，我们把当前进程叫做 P：

```
sleep_on (wait_queue_head_t *q)
{
         SLEEP_ON_VAR   ／＊宏定义，用来初始化要插入到等待队列中的元素＊／
         current->state = TASK_UNINTERRUPTIBLE;
         SLEEP_ON_HEAD  ／＊宏定义，把 P 插入到等待队列 ＊／
```

```
            schedule () ;
           SLEEP_ON_TAIL   /*宏定义把 P 从等待队列中删除 */
    }
```

这个函数把 P 的状态设置为 TASK_UNINTERRUPTIBLE，并把 P 插入等待队列。然后，它调用调度程序恢复另一个程序的执行。当 P 被唤醒时，调度程序恢复 sleep_on( )函数的执行，把 P 从等待队列中删除。

- interruptible_sleep_on( )与 sleep_on( )函数是一样的，但稍有不同，前者把进程 P 的状态设置为 TASK_INTERRUPTIBLE 而不是 TASK_UNINTERRUPTIBLE，因此，通过接受一个信号可以唤醒 P。
- sleep_on_timeout( ) 和 interruptible_sleep_on_timeout( )与前面情况类似，但它们允许调用者定义一个时间间隔，过了这个间隔以后，内核唤醒进程。为了做到这点，它们调用 schedule_timeout( )函数而不是 schedule( )函数。

利用 wake_up 或者 wake_up_interruptible 宏，让插入等待队列中的进程进入 TASK_RUNNING 状态，这两个宏最终都调用了 try_to_wake_up( )函数：

```
static inline int try_to_wake_up (struct task_struct * p, int synchronous)
{
     unsigned long flags;
       int success = 0;
      spin_lock_irqsave (&runqueue_lock, flags) ; /*加锁*/
       p->state = TASK_RUNNING;
    if  (task_on_runqueue (p) )      /*判断 p 是否已经在运行队列*/
            goto out;
       add_to_runqueue (p) ;        /*不在，则把 p 插入到运行队列*/
      if  (!synchronous || ! (p->cpus_allowed &  (1 << smp_processor_id () ) ) )   /
            reschedule_idle (p) ;
      success = 1;
  out:
    spin_unlock_irqrestore (&runqueue_lock, flags) ;  /*开锁*/
    return success;
}
```

在这个函数中，p 为要唤醒的进程。如果 p 不在运行队列中，则把它放入运行队列。如果重新调度正在进行的过程中，则调用 reschedule_idle () 函数，这个函数决定进程 p 是否应该抢占某一 CPU 上的当前进程（参见下一章）。

实际上，在内核的其他部分，最常用的还是 wake_up 或者 wake_up_interruptible 宏，也就是说，如果你要在内核级进行编程，只需调用其中的一个宏。例如一个简单的实时时钟（RTC）中断程序如下：

```
static DECLARE_WAIT_QUEUE_HEAD (rtc_wait) ; /*初始化等待队列首部*/

void rtc_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
        spin_lock (&rtc_lock) ;
        rtc_irq_data = CMOS_READ (RTC_INTR_FLAGS) ;
        spin_unlock (&rtc_lock) ;
        wake_up_interruptible (&rtc_wait) ;
}
```

这个中断处理程序通过从实时时钟的 I/O 端口（CMOS_READ 宏产生一对 outb/inb）读取

数据，然后唤醒在 rtc_wait 等待队列上睡眠的任务。

4.6　内核线程

内核线程（thread）或叫守护进程（daemon），在操作系统中占据相当大的比例，当 Linux 操作系统启动以后，尤其是 Xwindow 也启动以后，你可以用“ps”命令查看系统中的进程，这时会发现很多以“d”结尾的进程名，这些进程就是内核线程。

内核线程也可以叫内核任务，它们周期性地执行，例如，磁盘高速缓存的刷新，网络连接的维护，页面的换入换出等。在 Linux 中，内核线程与普通进程有一些本质的区别，从以下几个方面可以看出二者之间的差异。

- 内核线程执行的是内核中的函数，而普通进程只有通过系统调用才能执行内核中的函数。
- 内核线程只运行在内核态，而普通进程既可以运行在用户态，也可以运行在内核态。
- 因为内核线程指只运行在内核态，因此，它只能使用大于 PAGE_OFFSET（3G）的地址空间。另一方面，不管在用户态还是内核态，普通进程可以使用 4GB 的地址空间。

内核线程是由 kernel_thread(　)函数在内核态下创建的，这个函数所包含的代码大部分是内联式汇编语言，但在某种程度上等价于下面的代码：

```
int kernel_thread (int  (*fn) (void *) , void * arg,
                  unsigned long flags)
{
    pid_t p ;
    p = clone ( 0, flags | CLONE_VM ) ;
    if  ( p )         /* parent */
        return p;
    else {           /* child */
        fn (arg)  ;
        exit (  )  ;
   }
}
```

系统中大部分的内核线程是在系统的启动过程中建立的，其相关内容将在启动系统一章进行介绍。

4.7　进程的权能

Linux 用“权能（capability）”表示一进程所具有的权力。一种权能仅仅是一个标志，它表明是否允许进程执行一个特定的操作或一组特定的操作。这个模型不同于传统的“超级用户对普通用户”模型，在后一种模型中，一个进程要么能做任何事情，要么什么也不能做，这取决于它的有效 UID。也就是说，超级用户与普通用户的划分过于笼统。如表 4.13 给出了在 Linux 内核中已定义的权能。

表 4.13　　　　　　　　　　　　进程的权能

名字	描述
CAP_CHOWN	忽略对文件和组的拥有者进行改变的限制
CAP_DAC_OVERRIDE	忽略文件的访问许可权
CAP_DAC_READ_SEARCH	忽略文件/目录读和搜索的许可权

续表

名字	描述
CAP_FOWNER	忽略对文件拥有者的限制
CAP_FSETID	忽略对 setid 和 setgid 标志的限制
CAP_KILL	忽略对信号挂起的限制
CAP_SETGID	允许 setgid 标志的操作
CAP_SETUID	允许 setuid 标志的操作
CAP_SETPCAP	转移/删除对其他进程所许可的权能
CAP_LINUX_IMMUTABLE	允许对仅追加和不可变文件的修改
CAP_NET_BIND_SERVICE	允许捆绑到低于 1024TCP/UDP 的套节字
CAP_NET_BROADCAST	允许网络广播和监听多点传送
CAP_NET_ADMIN	允许一般的网络管理。
CAP_NET_RAW	允许使用 RAW 和 PACKET 套节字
CAP_IPC_LOCK	允许页和共享内存的加锁
CAP_IPC_OWNER	跳过 IPC 拥有者的检查
CAP_SYS_MODULE	允许内核模块的插入和删除
CAP_SYS_RAWIO	允许通过 ioperm( ) 和 iopl( )访问 I/O 端口
CAP_SYS_CHROOT	允许使用 chroot( )
CAP_SYS_PTRACE	允许在任何进程上使用 ptrace( )
CAP_SYS_PACCT	允许配置进程的计账
CAP_SYS_ADMIN	允许一般的系统管理
CAP_SYS_BOOT	允许使用 reboot( )
CAP_SYS_NICE	忽略对 nice( )的限制
CAP_SYS_RESOURCE	忽略对几个资源使用的限制
CAP_SYS_TIME	允许系统时钟和实时时钟的操作
CAP_SYS_TTY_CONFIG	允许配置 tty 设备

任何时候，每个进程只需要有限种权能，这是其主要优势。因此，即使一位有恶意的用户使用有潜在错误程序，他也只能非法地执行有限个操作类型。

例如，假定一个有潜在错误的程序只有 CAP_SYS_TIME 权能。在这种情况下，利用其错误的恶意用户只能在非法地改变实时时钟和系统时钟方面获得成功。他并不能执行其他任何特权的操作。

4.8　内核同步

内核中的很多操作在执行的过程中都不允许受到打扰，最典型的例子就是对队列的操作。如果两个进程都要将一个数据结构链入到同一个队列的尾部，要是在第 1 个进程完成了一半的时候发生了调度，让第 2 个进程插了进来，就可能造成混乱。类似的干扰可能来自某个中断服务程序或 bh 函数。在多处理机 SMP 结构的系统中，这种干扰还有可能来自另一个处理器。这种干扰本质上表项为对临界资源（如队列）的互斥使用。下面介绍几种避免这种干扰的同步方式。

4.8.1　信号量

进程间对共享资源的互斥访问是通过 “ 信号量 ” 机制来实现的。信号量机制是操作系统教科书中比较重要的内容之一。Linux 内核中提供了两个函数 down()和 up（），分别对应于操作系统教科书中的 P、V 操作。

信号量在内核中定义为 semaphore 数据结构，位于 include/i386/semaphore.h：

```
struct semaphore {
        atomic_t count;
            int sleepers;
        wait_queue_head_t wait;
 #if WAITQUEUE_DEBUG
        long __magic;
 #endif
 };
```

其中的 count 域就是 “ 信号量 ” 中的那个 “ 量 ”，它代表着可用资源的数量。如果该值大于 0，那么资源就是空闲的，也就是说，该资源可以使用。相反，如果 count 小于 0，那么这个信号量就是繁忙的，也就是说，这个受保护的资源现在不能使用。在后一种情况下，count 的绝对值表示了正在等待这个资源的进程数。该值为 0 表示有一个进程正在使用这个资源，但没有其他进程在等待这个资源。

Wait 域存放等待链表的地址，该链表中包含正在等待这个资源的所有睡眠的进程。当然，如果 count 大于或等于 0，则等待队列为空。为了明确表示等待队列中正在等待的进程数，引入了计数器 sleepers。

down()和 up()函数主要应用在文件系统和驱动程序中，把要保护的临界区放在这两个函数中间，用法如下：

```
down（）;
临界区
up（）;
```

这两个函数是用嵌入式汇编实现的，非常麻烦，在此不予详细介绍。

4.8.2　原子操作

避免干扰的最简单方法就是保证操作的原子性，即操作必须在一条单独的指令内执行。

有两种类型的原子操作，即位图操作和数学的加减操作。

1．位图操作

在内核的很多地方用到位图，例如内存管理中对空闲页的管理，位图还有一个广泛的用途就是简单的加锁，例如提供对打开设备的互斥访问。关于位图的操作函数如下：

以下函数的参数中，addr 指向位图。

- void set_bit(int nr, volatile void *addr)：设置位图的第 nr 位。
- void clear_bit(int nr, volatile void *addr): 清位图的第 nr 位。
- void change_bit(int nr, volatile void *addr): 改变位图的第 nr 位。
- int test_and_set_bit(int nr, volatile void *addr): 设置第 nr 位，并返回该位原来的值，且两个操作是原子操作，不可分割。
- int test_and_clear_bit(int nr, volatile void *addr): 清第 nr 为，并返回该位原来的值，且两个操作是原子操作。
- int test_and_change_bit(int nr, volatile void *addr)：改变第 nr 位，并返回该位原来的值，且这两个操作是原子操作。

这些操作利用了 LOCK_PREFIX 宏，对于 SMP 内核，该宏是总线锁指令的前缀，对于单 CPU 这个宏不起任何作用。这就保证了在 SMP 环境下访问的原子性。

2．算术操作

有时候位操作是不方便的，取而代之的是需要执行算术操作，即加、减操作及加 1、减 1 操作。典型的例子是很多数据结构中的引用计数域 count（如 inode 结构）。这些操作的原子性是由 atomic_t 数据类型和表 4.14 中的函数保证的。atomic_t 的类型在 include/i386/atomic.h，定义如下：

```
typedef struct { volatile int counter; } atomic_t;
```

表 4.14　　原子操作

函数	说明
atomic_read(v)	返回*v
atomic_set(v,i)	把*v 设置成 I
Atomic_add(i,v)	给*v 增加 I
Atomic_sub(i,v)	从*v 中减去 I
Atomic_inc(v)	给*v 加 1
Atomic_dec(v)	从*v 中减去 1
Atomic_dec_and_test(v)	从*v 中减去 1，如果结果非空就返回 1；否则返回 0
Atomic_inc_and_test_greater_zero(v)	给*v 加 1，如果结果为正就返回 1；否则就返回 0
Atomic_clear_mask(mask,addr)	清除由 mask 所指定的 addr 中的所有位
Atomic_set_mask(mask,addr)	设置由 mask 所指定的 addr 中的所有位

4.8.3 自旋锁、读写自旋锁和大读者自旋锁

在 Linux 开发的早期，开发者就面临这样的问题，即不同类型的上下文（用户进程对中断）如何访问共享的数据，以及如何访问来自多个 CPU 同一上下文的不同实例。

在 Linux 内核中，临界区的代码或者是由进程上下文来执行，或者是由中断上下文来执行。在单 CPU 上，可以用 cli/sti 指令来保护临界区的使用，例如：

```
unsigned long flags;

save_flags(flags);
cli();
/* critical code */
restore_flags(flags);
```

但是，在 SMP 上，这种方法明显是没有用的，因为同一段代码序列可能由另一个进程同时执行,而 cli()仅能单独地为每个 CPU 上的中断上下文提供对竞争资源的保护，它无法对运行在不同 CPU 上的上下文提供对竞争资源的访问。因此，必须用到自旋锁。

所谓自旋锁，就是当一个进程发现锁被另一个进程锁着时，它就不停地“旋转”，不断执行一个指令的循环直到锁打开。自旋锁只对 SMP 有用，对单 CPU 没有意义。

有 3 种类型的自旋锁：基本的、读写以及大读者自旋锁。读写自旋锁适用于“多个读者少数写者”的场合，例如，有多个读者仅有一个写者，或者没有读者只有一个写者。大读者自旋锁是读写自旋锁的一种，但更照顾读者。大读者自旋锁现在主要用在 Sparc64 和网络系统中。

本章对进程进行了全面描述，但并不是对每个部分都进行了深入描述，因为在整个操作系统中，进程处于核心位置，因此，内核的其他部分（如文件、内存、进程间的通信等）都与进程有密切的联系，相关内容只能在后续的章节中涉及到。通过本章的介绍，读者应该对进程有一个全方位的认识：

（1）进程是由正文段（Text）、用户数据段（User Segment）以及系统数据段（System Segment）共同组成的一个执行环境。

（2）Linux 中用 task_struct 结构来描述进程，也就是说，有关进程的所有信息都存储在这个数据结构中，或者说，Linux 中的进程与 task_struct 结构是同意词，在英文描述中，有时把进程（Process）和线程（Thread）混在一起使用，但并不是说，进程与线程有同样的含义，只不过描述线程的数据结构也是 task_struct。task_struct 就是一般教科书上所讲的进程控制块.。

（3）本章对 task_struct 结构中存放的信息进行了分类讨论，但并不要求在此能掌握所有的内容，相对独立的内容为进程的状态，在此再次给出概述。

- TASK_RUNNING：也就是通常所说的就绪（Ready）状态。
- TASK_INTERRUPTIBLE：等待一个信号或一个资源（睡眠状态）。
- TASK_UNINTERRUPTIBLE：等待一个资源（睡眠状态），处于某个等待队列中。
- TASK_ZOMBIE：没有父进程的子进程。
- TASK_STOPPED：正在被调试的任务。

（4）task_struct 结构与内核栈存放在一起，占 8KB 的空间。

（5）当前进程就是在某个 CPU 上正在运行的进程，Linux 中用宏 current 来描述，也可以把 curennt 当作一个全局变量来用。

（6）为了把内核中的所有进程组织起来，Linux 提供了几种组织方式，其中哈希表和双向循环链表方式是针对系统中的所有进程（包括内核线程），而运行队列和等待队列是把处于同一状态的进程组织起来。

（7）Linux 2.4 中引入一种通用链表 list_head，这是面向对象思想在 C 中的具体实现，在内核中其他使用链表的地方都引用了这种基类型。

（8）进程的权能和内核的同步我们仅仅做了简单介绍，因为进程管理会涉及到这些内容，但它们不是进程管理的核心内容，引入这些内容仅仅是为了让读者在阅读源代码时扫除一些障碍。

（9）进程的创建及执行将在第六章的最后一节进行讨论。

第五章　进程调度与切换

在多进程的操作系统中，进程调度是一个全局性、关键性的问题，它对系统的总体设计、系统的的实现、功能设置以及各个方面的性能都有着决定性的影响。根据调度的结果所作的进程切换的速度，也是衡量一个操作系统性能的重要指标。进程调度算法的设计，还对系统的复杂性有着极大的影响，常常会由于实现的复杂程度而在功能与性能方面作出必要的权衡和让步。

进程调度与时间的关系非常密切，因此，本章首先讨论与时间相关的主题，然后才讨论进程的调度，最后介绍了 Linux 中进程是如何进行切换的。

5.1　Linux 时间系统

计算机是以严格精确的时间进行数值运算和和数据处理的，最基本的时间单元是时钟周期，例如取指令、执行指令、存取内存等，但是我们不讨论这些纯硬件的东西，这里要讨论的是操作系统建立的时间系统，这个时间系统是整个操作系统活动的动力。

时间系统是计算机系统非常重要的组成部分，特别是对于 UNIX 类分时系统尤为重要。时间系统通常又被简称为时钟，它的主要任务是维持系统时间并且防止某个进程独占 CPU 及其他资源，也就是驱动进程的调度。本节将详细讲述时钟的来源、在 Linux 中的实现及其重要作用，使读者消除对时钟的神秘感。

5.1.1　时钟硬件

大部分 PC 机中有两个时钟源，他们分别叫做 RTC 和 OS（操作系统）时钟。RTC（Real Time Clock，实时时钟）也叫做 CMOS 时钟，它是 PC 主机板上的一块芯片（或者叫做时钟电路），它靠电池供电，即使系统断电，也可以维持日期和时间。由于它独立于操作系统，所以也被称为硬件时钟，它为整个计算机提供一个计时标准，是最原始最底层的时钟数据。

Linux 只用 RTC 来获得时间和日期，同时，通过作用于/dev/rtc 设备文件，也允许进程对 RTC 编程。内核通过 0x70 和 0x71 I/O 端口存取 RTC。通过执行/sbin/clock 系统程序（它直接作用于这两个 I/O 端口），系统管理员可以配置时钟。

OS 时钟产生于 PC 主板上的定时/计数芯片，由操作系统控制这个芯片的工作，OS 时钟的基本单位就是该芯片的计数周期。在开机时操作系统取得 RTC 中的时间数据来初始化 OS 时钟，然后通过计数芯片的向下计数形成了 OS 时钟，所以 OS 时钟并不是本质意义上的时钟，它更应该被称为一个计数器。OS 时钟只在开机时才有效，而且完全由操作系统控制，所以也

被称为软时钟或系统时钟。下面我们重点描述 OS 时钟的产生。

OS 时钟所用的定时/计数芯片最典型的是 8253/8254 可编程定时/计数芯片，其硬件结构及工作原理在这里不详细讲述，只简单地描述它是怎样维持 OS 时钟的。OS 时钟的物理产生示意图如图 5.1 所示。

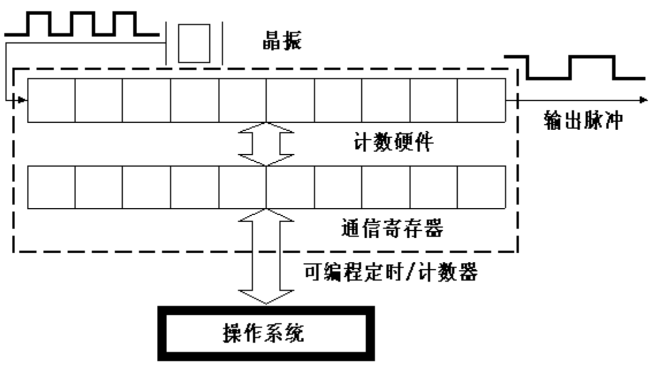


图 5.1　8253/8254 工作示意图

可编程定时/计数器总体上由两部分组成：计数硬件和通信寄存器。通信寄存器包含有控制寄存器、状态寄存器、计数初始值寄存器（16 位）、计数输出寄存器等。通信寄存器在计数硬件和操作系统之间建立联系，用于二者之间的通信，操作系统通过这些寄存器控制计数硬件的工作方式、读取计数硬件的当前状态和计数值等信息。在 Linux 内核初始化时，内核写入控制字和计数初值，这样计数硬件就会按照一定的计数方式对晶振产生的输入脉冲信号（5MHz~100MHz 的频率）进行计数操作：计数器从计数初值开始，每收到一次脉冲信号，计数器减 1，当计数器减至 0 时，就会输出高电平或低电平，然后，如果计数为循环方式（通常为循环计数方式），则重新从计数初值进行计数，从而产生如图 5.1 所示的输出脉冲信号（当然不一定是很规则的方波）。这个输出脉冲是 OS 时钟的硬件基础，之所以这么说，是因为这个输出脉冲将接到中断控制器上 ，产生中断信号，触发后面要讲的时钟中断，由时钟中断服务程序维持 OS 时钟的正常工作，所谓维持，其实就是简单的加 1 及细微的修正操作。这就是 OS 时钟产生的来源。

5.12　时钟运作机制

不同的操作系统，RTC 和 OS 时钟的关系是不同的。RTC 和 OS 时钟之间的关系通常也被称作操作系统的时钟运作机制。

一般来说，RTC 是 OS 时钟的时间基准，操作系统通过读取 RTC 来初始化 OS 时钟，此后二者保持同步运行，共同维持着系统时间。保持同步运行是什么意思呢？就是指操作系统运行过程中，每隔一个固定时间会刷新或校正 RTC 中的信息。

Linux 中的时钟运作机制如图 5.2 所示。OS 时钟和 RTC 之间要通过 BIOS 的连接，是因为传统 PC 机的 BIOS 中固化有对 RTC 进行有关操作的函数，例如 INT 1AH 等中断服务程序，

通常操作系统也直接利用这些函数对 RTC 进行操作,例如从 RTC 中读出有关数据对 OS 时钟初始化、对 RTC 进行更新等。实际上，不通过 BIOS 而直接对 RTC 的有关端口进行操作也是可以的。Linux 中在内核初始化完成后就完全抛弃了 BIOS 中的程序。

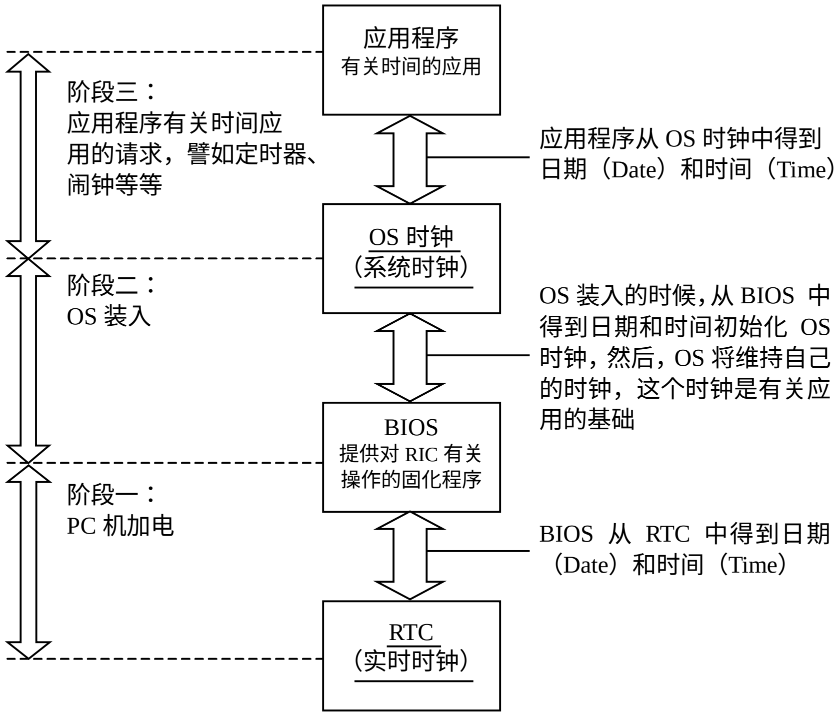


图 5.2　时钟运作机制

我们可以看到，RTC 处于最底层，提供最原始的时钟数据。OS 时钟建立在 RTC 之上，初始化完成后将完全由操作系统控制，和 RTC 脱离关系。操作系统通过 OS 时钟提供给应用程序所有和时间有关的服务。因为 OS 时钟完全是一个软件问题，其所能表达的时间由操作系统的设计者决定，将 OS 时钟定义为整型还是长整型或者大的超乎想像都是设计者的事。

5.1.3　Linux 时间基准

以上我们了解了 RTC（实时时钟、硬件时钟）和 OS 时钟（系统时钟、软时钟）。下面我们具体描述 OS 时钟。

我们知道，OS 时钟是由可编程定时/计数器产生的输出脉冲触发中断而产生的。输出脉冲的周期叫做一个“时钟滴答”，有些书上也把它叫做“时标”。计算机中的时间是以时钟滴答为单位的，每一次时钟滴答，系统时间就会加 1。操作系统根据当前时钟滴答的数目就可以得到以秒或毫秒等为单位的其他时间格式。

不同的操作系统采用不同的“时间基准”。定义“时间基准”的目的是为了简化计算，这样计算机中的时间只要表示为从这个时间基准开始的时钟滴答数就可以了。“时间基准”是由操作系统的设计者规定的。例如 DOS 的时间基准是 1980 年 1 月 1 日，UNIX 和 Minux 的

时间基准是 1970 年 1 月 1 日上午 12 点，Linux 的时间基准是 1970 年 1 月 1 日凌晨 0 点。

5.1.4 Linux 的时间系统

通过上面的时钟运作机制，我们知道了 OS 时钟在 Linux 中的重要地位。OS 时钟记录的时间也就是通常所说的系统时间。系统时间是以“时钟滴答”为单位的，而时钟中断的频率决定了一个时钟滴答的长短，例如每秒有 100 次时钟中断，那么一个时钟滴答的就是 10 毫秒（记为 10ms），相应地，系统时间就会每 10ms 增 1。不同的操作系统对时钟滴答的定义是不同的，例如 DOS 的时钟滴答为 1/18.2s，Minix 的时钟滴答为 1/60s 等。

Linux 中用全局变量 jiffies 表示系统自启动以来的时钟滴答数目。jiffy 是“瞬间、一会儿”的意思，和“时钟滴答”表达的是同一个意思。jiffies 是 jiffy 的复数形式，在 /kernel/time.c 中定义如下：

```
unsigned long volatile jiffies
```

在 jiffies 基础上，Linux 提供了如下适合人们习惯的时间格式，在 /include/linux/time.h 中定义如下：

```
struct timespec {                /*  这是精度很高的表示*/
     long tv_sec;             /* 秒  (second)  */
     long tv_nsec;            /* 纳秒：十亿分之一秒（ nanosecond）*/
};

struct timeval  {                /* 普通精度    */
     int  tv_sec;             /*  秒    */
     int  tv_usec;            /* 微秒：百万分之一秒（microsecond）*/
};

struct timezone {                      /*  时区  */
     int  tz_minuteswest;          /*  格林尼治时间往西方的时差 */
     int  tz_dsttime;                 /* 时间修正方式 */
};
```

tv_sec 表示秒（second），tv_usec 表示微秒（microsecond，百万分之一秒即 10^{-6} 秒），tv_nsec 表示纳秒（nanosecond，十亿分之一秒即 10^{-9} 秒）。定义 tb_usec 和 tv_nsec 的目的是为了适用不同的使用要求，不同的场合根据对时间精度的要求选用这两种表示。

另外，Linux 还定义了用于表示更加符合大众习惯的时间表示：年、月、日。但是万变不离其宗，所有的时间应用都是建立在 jiffies 基础之上的，我们将详细讨论 jiffies 的产生和其作用。简而言之，jiffies 产生于时钟中断！

5.2　时钟中断

5.2.1　时钟中断的产生

前面我们看到，Linux 的 OS 时钟的物理产生原因是可编程定时/计数器产生的输出脉冲，这个脉冲送入 CPU，就可以引发一个中断请求信号，我们就把它叫做时钟中断。

“时钟中断”是特别重要的一个中断，因为整个操作系统的活动都受到它的激励。系统利用时钟中断维持系统时间、促使环境的切换，以保证所有进程共享 CPU；利用时钟中断进行记帐、监督系统工作以及确定未来的调度优先级等工作。可以说，“时钟中断”是整个操作系统的脉搏。

时钟中断的物理产生如图 5.3 所示。

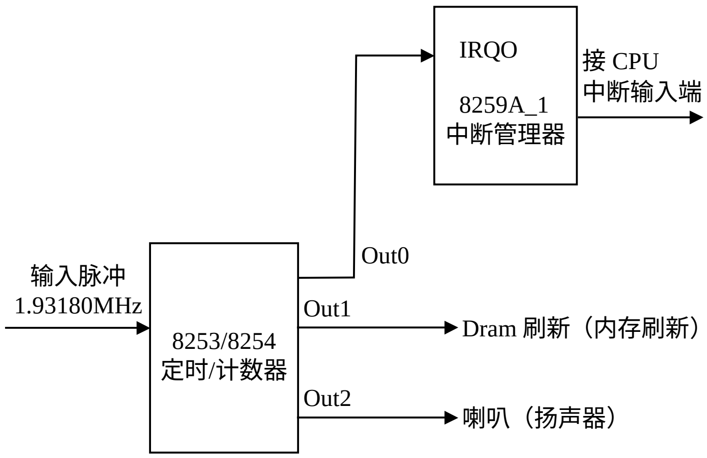


图 5.3　8253 和 8259A 的物理连接方式

操作系统对可编程定时/计数器进行有关初始化，然后定时/计数器就对输入脉冲进行计数（分频），产生的 3 个输出脉冲：Out0、Out1、Out2，它们各有用途，很多有关接口的书都介绍了这个问题，我们只介绍 Out0 上的输出脉冲，这个脉冲信号接到中断控制器 8259A_1 的 0 号管脚，触发一个周期性的中断，我们就把这个中断叫做时钟中断，时钟中断的周期，也就是脉冲信号的周期，我们叫做“滴答”或“时标”（tick）。从本质上说，时钟中断只是一个周期性的信号，完全是硬件行为，该信号触发 CPU 去执行一个中断服务程序，但是为了方便，我们就把这个服务程序叫做时钟中断，读者可能早就习惯这种叫法了，我们也不必把一些概念区分得那么详细。

5.2.2 Linux 实现时钟中断的全过程

1．可编程定时/计数器的初始化

IBM PC 中使用的是 8253 或 8254 芯片。有关该芯片的详细知识我们不再详述，只大体介绍以下它的组成和作用，如表 5.1 所示。

表 5.1　　8253/8254 的组成及作用

名称	端口地址	工作方式	产生的输出脉冲的用途
计数器 0	0x40	方式 3	时钟中断，也叫系统时钟
计数器 1	0x41	方式 2	动态存储器刷新
计数器 2	0x42	方式 3	扬声器发声
控制寄存器	0x43	/	用于 8253 的初始化，接收控制字

计数器 0 的输出就是图 5.3 中的 Out0,它的频率由操作系统的设计者确定,Linux 对 8253 的初始化程序段如下（在/arch/i386/kernel/i8259.c 的 init_IRQ()函数中）：

```
set_intr_gate（0x20, interrupt[0]）;
```

／*在 IDT 的第 0x20 个表项中插入一个中断门。这个门中的段选择符设置成内核代码段的选择符，偏移域设置成 0 号中断处理程序的入口地址。*/

```
outb_p（0x34,0x43）;          /* 写计数器 0 的控制字：工作方式 2*/
outb_p（LATCH & 0xff , 0x40）;    /* 写计数初值 LSB  计数初值低位字节*/
outb（LATCH >> 8 , 0x40）;   /* 写计数初值 MSB 计数初值高位字节*/
```

LATCH（英文意思为：锁存器，即其中锁存了计数器 0 的初值）为计数器 0 的计数初值，在/include/linux/timex.h 中定义如下：

```
#define CLOCK_TICK_RATE 1193180     /* 图 5.3 中的输入脉冲 */
#define LATCH   （（CLOCK_TICK_RATE + HZ/2） / HZ）   /* 计数器 0 的计数初值 */
```

CLOCK_TICK_RATE 是整个 8253 的输入脉冲，如图 5.3 中所示为 1.193180MHz，是近似为 1MHz 的方波信号,8253 内部的 3 个计数器都对这个时钟进行计数,进而产生不同的输出信号，用于不同的用途。

HZ 表示计数器 0 的频率，也就是时钟中断或系统时钟的频率，在/include/asm/param.h 中定义如下:

```
#define HZ 100
```

2．与时钟中断相关的函数

下面我们接着介绍时钟中断触发的服务程序，该程序代码比较复杂，分布在不同的源文件中，主要包括如下函数：

- 时钟中断程序：timer_interrupt（ ）；

- 中断服务通用例程：do_timer_interrupt();
- 时钟函数：do_timer（）；
- 中断安装程序：setup_irq（）；
- 中断返回函数：ret_from_intr（）；

前 3 个函数的调用关系如下：

```
timer_interrupt（）
        ↓
    do_timer_interrupt()
              ↓
         do_timer（）
```

(1) timer_interrupt（）

这个函数大约每 10ms 被调用一次，实际上，timer_interrupt（）函数是一个封装例程，它真正做的事情并不多，但是，作为一个中断程序,它必须在关中断的情况下执行。如果只考虑单处理机的情况，该函数主要语句就是调用 do_timer_interrupt()函数。

(2) do_timer_interrupt()

do_timer_interrupt()函数有两个主要任务，一个是调用 do_timer（），另一个是维持实时时钟（RTC，每隔一定时间段要回写），其实现代码在/arch/i386/kernel/time.c 中，为了突出主题，笔者对以下函数作了改写，以便于读者理解：

```
static inline void do_timer_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
     do_timer (regs) ; /* 调用时钟函数，将时钟函数等同于时钟中断未尝不可*/

     if (xtime.tv_sec > last_rtc_update + 660)
update_RTC();
/*每隔 11 分钟就更新 RTC 中的时间信息，以使 OS 时钟和 RTC 时钟保持同步，11 分钟即 660 秒，xtime.tv_sec 的单位是秒，last_rtc_update 记录的是上次 RTC 更新时的值 */
}
```

其中，xtime 是前面所提到的 timeval 类型，这是一个全局变量。

(3) 时钟函数 do_timer() （在/kernel/sched.c 中）

```
void do_timer (struct pt_regs * regs)
{
      (* (unsigned long *) &jiffies) ++;  /*更新系统时间，这种写法保证对 jiffies 操作的原子性*/
 update_process_times();
     ++lost_ticks;
     if ( ! user_mode  ( regs )  )
          ++lost_ticks_system;

          mark_bh (TIMER_BH) ;
     if  (tq_timer)
          mark_bh (TQUEUE_BH) ;
}
```

其中，update_process_times()函数与进程调度有关，从函数的名子可以看出，它处理

的是与当前进程与时间有关的变量，例如，要更新当前进程的时间片计数器 counter，如果 counter<=0，则要调用调度程序，要处理进程的所有定时器：实时、虚拟、概况，另外还要做一些统计工作。

与时间有关的事情很多，不能全都让这个函数去完成，这是因为这个函数是在关中断的情况下执行，必须处理完最重要的时间信息后退出，以处理其他事情。那么，与时间相关的其他信息谁去处理，何时处理？这就是由第三章讨论的后半部分去去处理。上面 timer_interrupt()（包括它所调用的函数）所做的事情就是上半部分。

在该函数中还有两个变量 lost_ticks 和 lost_ticks_system,这是用来记录 timer_bh()执行前时钟中断发生的次数。因为时钟中断发生的频率很高(每 10ms 一次),所以在 timer_bh()执行之前，可能已经有时钟中断发生了，而 timer_bh()要提供定时、计费等重要操作，所以为了保证时间计量的准确性，使用了这两个变量。lost_ticks 用来记录 timer_bh()执行前时钟中断发生的次数，如果时钟中断发生时当前进程运行于内核态，则 lost_ticks_system 用来记录 timer_bh()执行前在内核态发生时钟中断的次数，这样可以对当前进程精确计费。

（4）中断安装程序

从上面的介绍可以看出，时钟中断与进程调度密不可分，因此，一旦开始有时钟中断就可能要进行调度，在系统进行初始化时，所做的大量工作之一就是对时钟进行初始化，其函数 time_init ()的代码在/arch/i386/kernel/time.c 中，对其简写如下：

```
 void __init time_init (void)
  {
xtime.tv_sec=get_cmos_time();
xtime.tv_usec=0;
setup_irq (0, &irq0) ;
}
```

其中的 get_cmos_time()函数就是把当时的实际时间从 CMOS 时钟芯片读入变量 xtime 中，时间精度为秒。而 setup_irq (0, &irq0) 就是时钟中断安装函数，那么 irq0 指的是什么呢，它是一个结构类型 irqaction，其定义及初值如下：

```
static struct irqaction irq0  = { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
```

setup_irq (0, &irq0) 的代码在/arch/i386/kernel/irq.c 中，其主要功能就是将中断程序连入相应的中断请求队列，以等待中断到来时相应的中断程序被执行。

到现在为止，我们仅仅是把时钟中断程序挂入中断请求队列，什么时候执行，怎样执行，这是一个复杂的过程（参见第三章），为了让读者对时钟中断有一个完整的认识，我们忽略中间过程，而给出一个整体描述。我们将有关函数改写如下，体现时钟中断的大意：

```
do_timer_interrupt ( )          /*这是一个伪函数 */
{
    SAVE_ALL                    /*保存处理机现场 */
    intr_count += 1;              /* 这段操作不允许被中断 */
    timer_interrupt()             /* 调用时钟中断程序 */
    intr_count -= 1;
    jmp ret_from_intr             /* 中断返回函数 */
}
```

其中，jmp ret_from_intr 是一段汇编代码，也是一个较为复杂的过程，它最终要调用 jmp ret_from_sys_call，即系统调用返回函数，而这个函数与进程的调度又密切相关，因此，

我们重点分析 jmp ret_from_sys_call。

3．系统调用返回函数

系统调用返回函数的源代码在/arch/i386/kernel/entry.S 中

```
ENTRY (ret_from_sys_call)
        cli                 # need_resched and signals atomic test
        cmpl $0,need_resched (%ebx)
        jne reschedule
        cmpl $0,sigpending (%ebx)
        jne signal_return
 restore_all:
        RESTORE_ALL

        ALIGN
 signal_return:
        sti              # we can get here from an interrupt handler
        testl $ (VM_MASK) ,EFLAGS (%esp)
        movl %esp,%eax
        jne v86_signal_return
        xorl %edx,%edx
        call SYMBOL_NAME (do_signal)
        jmp restore_all

        ALIGN
       v86_signal_return:
        call SYMBOL_NAME (save_v86_state)
        movl %eax,%esp
        xorl %edx,%edx
        call SYMBOL_NAME (do_signal)
        jmp restore_all
  ….
 reschedule:
        call SYMBOL_NAME (schedule)     # test
       jmp ret_from_sys_call
```

这一段汇编代码就是前面我们所说的“从系统调用返回函数”ret_from_sys_call，它是从中断、异常及系统调用返回时的通用接口。这段代码主体就是 ret_from_sys_call 函数，其执行过程中要调用其他一些函数（实际上是一段代码，不是真正的函数），在此我们列出相关的几个函数。

（1）ret_from_sys_call：主体。

（2）reschedule：检测是否需要重新调度。

（3）signal_return：处理当前进程接收到的信号。

（4）v86_signal_return：处理虚拟 86 模式下当前进程接收到的信号。

（5）RESTORE_ALL：我们把这个函数叫做彻底返回函数，因为执行该函数之后，就返回到当前进程的地址空间中去了。

可以看到 ret_from_sys_call 的主要作用有：检测调度标志 need_resched，决定是否要执行调度程序；处理当前进程的信号；恢复当前进程的环境使之继续执行。

最后我们再次从总体上浏览一下时钟中断：

每个时钟滴答，时钟中断得到执行。时钟中断执行的频率很高：100 次/秒，时钟中断的主要工作是处理和时间有关的所有信息、决定是否执行调度程序以及处理下半部分。和时间有关的所有信息包括系统时间、进程的时间片、延时、使用 CPU 的时间、各种定时器，进程更新后的时间片为进程调度提供依据，然后在时钟中断返回时决定是否要执行调度程序。下半部分处理程序是 Linux 提供的一种机制，它使一部分工作推迟执行。时钟中断要绝对保证维持系统时间的准确性，而下半部分这种机制的提供不但保证了这种准确性，还大幅提高了系统性能。

5.3 Linux 的调度程序——Schedule（ ）

进程的合理调度是一个非常复杂的工作，它取决于可执行程序的类型（实时或普通）、调度的策略及操作系统所追求的目标，幸运的是，Linux 的调度程序比较简单。

5.3.1 基本原理

从前面我们可以看到，进程运行需要各种各样的系统资源，如内存、文件、打印机和最宝贵的 CPU 等，所以说，调度的实质就是资源的分配。系统通过不同的调度算法（Scheduling Algorithm）来实现这种资源的分配。通常来说，选择什么样的调度算法取决于资源分配的策略（Scheduling Policy），我们不准备在这里详细说明各种调度算法，只说明与 Linux 调度相关的几种算法及这些算法的原理。

一个好的调度算法应当考虑以下几个方面。

（1）公平：保证每个进程得到合理的 CPU 时间。

（2）高效：使 CPU 保持忙碌状态，即总是有进程在 CPU 上运行。

（3）响应时间：使交互用户的响应时间尽可能短。

（4）周转时间：使批处理用户等待输出的时间尽可能短。

（5）吞吐量：使单位时间内处理的进程数量尽可能多。

很显然，这 5 个目标不可能同时达到，所以，不同的操作系统会在这几个方面中作出相应的取舍，从而确定自己的调度算法，例如 UNIX 采用动态优先数调度、5.3BSD 采用多级反馈队列调度、Windows 采用抢先多任务调度等。

下面来了解一下主要的调度算法及其基本原理。

1．时间片轮转调度算法

时间片（Time Slice）就是分配给进程运行的一段时间。

在分时系统中，为了保证人机交互的及时性，系统使每个进程依次地按时间片轮流的方式执行，此时即应采用时间片轮转法进行调度。在通常的轮转法中，系统将所有的可运行（即就绪）进程按先来先服务的原则，排成一个队列，每次调度时把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms 不等。当执行的时间片用完时，系统发

出信号，通知调度程序，调度程序便据此信号来停止该进程的执行，并将它送到运行队列的末尾，等待下一次执行。然后，把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证运行队列中的所有进程，在一个给定的时间（人所能接受的等待时间）内，均能获得一时间片的处理机执行时间。

2．优先权调度算法

为了照顾到紧迫型进程在进入系统后便能获得优先处理，引入了最高优先权调度算法。当将该算法用于进程调度时，系统将把处理机分配给运行队列中优先权最高的进程，这时，又可进一步把该算法分成两种方式。

（1）非抢占式优先权算法（又称不可剥夺调度，Nonpreemptive Scheduling）

在这种方式下，系统一旦将处理机（CPU）分配给运行队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可将处理机分配给另一个优先权高的进程。这种调度算法主要用于批处理系统中，也可用于某些对实时性要求不严的实时系统中。

（2）抢占式优先权调度算法（又称可剥夺调度，Preemptive Scheduling）

该算法的本质就是系统中当前运行的进程永远是可运行进程中优先权最高的那个。

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但是只要一出现了另一个优先权更高的进程时，调度程序就暂停原最高优先权进程的执行，而将处理机分配给新出现的优先权最高的进程，即剥夺当前进程的运行。因此，在采用这种调度算法时，每当出现一新的可运行进程，就将它和当前运行进程进行优先权比较，如果高于当前进程，将触发进程调度。

这种方式的优先权调度算法，能更好的满足紧迫进程的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。Linux 也采用这种调度算法。

3．多级反馈队列调度

这是时下最时髦的一种调度算法。其本质是：综合了时间片轮转调度和抢占式优先权调度的优点，即：优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片。

4．实时调度

最后我们来看一下实时系统中的调度。什么叫实时系统，就是系统对外部事件有求必应、尽快响应。在实时系统中存在有若干个实时进程或任务，它们用来反应或控制某个（些）外部事件，往往带有某种程度的紧迫性，因而对实时系统中的进程调度有某些特殊要求。

在实时系统中，广泛采用抢占调度方式，特别是对于那些要求严格的实时系统。因为这种调度方式既具有较大的灵活性，又能获得很小的调度延迟；但是这种调度方式也比较复杂。

我们大致了解以上的调度方式以后，下面具体来看 Linux 中的调度程序，这里要说明的是，Linux 的调度程序并不复杂，但这并不影响 Linux 调度程序的高效性！

5.3.2 Linux 进程调度时机

调度程序虽然特别重要，但它不过是一个存在于内核空间中的函数而已，并不神秘。Linux 的调度程序是一个叫 Schedule()的函数，这个函数被调用的频率很高，由它来决定是否要进行进程的切换，如果要切换的话，切换到哪个进程等。我们先来看在什么情况下要执行调度程序，我们把这种情况叫做调度时机。

Linux 调度时机主要有。

（1）进程状态转换的时刻：进程终止、进程睡眠；

（2）当前进程的时间片用完时（current->counter=0）；

（3）设备驱动程序；

（4）进程从中断、异常及系统调用返回到用户态时。

时机 1，进程要调用 sleep()或 exit()等函数进行状态转换，这些函数会主动调用调度程序进行进程调度。

时机 2，由于进程的时间片是由时钟中断来更新的，因此，这种情况和时机 4 是一样的。

时机 3，当设备驱动程序执行长而重复的任务时，直接调用调度程序。在每次反复循环中，驱动程序都检查 need_resched 的值，如果必要，则调用调度程序 schedule()主动放弃 CPU。

时机 4，如前所述，不管是从中断、异常还是系统调用返回，最终都调用 ret_from_sys_call()，由这个函数进行调度标志的检测，如果必要，则调用调用调度程序。那么，为什么从系统调用返回时要调用调度程序呢？这当然是从效率考虑。从系统调用返回意味着要离开内核态而返回到用户态，而状态的转换要花费一定的时间，因此，在返回到用户态前，系统把在内核态该处理的事全部做完。

对于直接执行调度程序的时机，我们不讨论，因为后面我们将描述调度程序的工作过程。前面我们讨论了时钟中断，知道了时钟中断的重要作用，下面我们就简单看一下每个时钟中断发生时内核要做的工作，首先对这个最频繁的调度时机有一个大体了解，然后再详细讨论调度程序的具体工作过程。

每个时钟中断（timer interrupt）发生时，由 3 个函数协同工作，共同完成进程的选择和切换，它们是：schedule()、do_timer()及 ret_form_sys_call()。我们先来解释一下这 3 个函数。

- schedule()：进程调度函数，由它来完成进程的选择（调度）。
- do_timer()：暂且称之为时钟函数，该函数在时钟中断服务程序中被调用，是时钟中断服务程序的主要组成部分，该函数被调用的频率就是时钟中断的频率即每秒钟 100 次（简称 100 赫兹或 100Hz）；
- ret_from_sys_call()：系统调用返回函数。当一个系统调用或中断完成时，该函数被调用，用于处理一些收尾工作，例如信号处理、核心任务等。

这 3 个函数是如何协调工作的呢？

前面我们讲过，时钟中断是一个中断服务程序，它的主要组成部分就是时钟函数 do_timer()，由这个函数完成系统时间的更新、进程时间片的更新等工作，更新后的进程时间片 counter 作为调度的主要依据。

在时钟中断返回时，要调用函数 ret_from_sys_call()，前面我们已经讨论过这个函数，在这个函数中有如下几行：

```
cmpl $0, _need_resched
jne reschedule
     ……
restore_all:
     RESTORE_ALL

reschedule:
     call SYMBOL_NAME (schedule)
     jmp ret_from_sys_call
```

这几行的意思很明显：检测 need_resched 标志，如果此标志为非 0，那么就转到 reschedule 处调用调度程序 schedule()进行进程的选择。调度程序 schedule()会根据具体的标准在运行队列中选择下一个应该运行的进程。当从调度程序返回时，如果发现又有调度标志被设置，则又调用调度程序，直到调度标志为 0，这时，从调度程序返回时由 RESTORE_ALL 恢复被选定进程的环境，返回到被选定进程的用户空间，使之得到运行。

以上就是时钟中断这个最频繁的调度时机。讨论这个的主要目的使读者对时机 4 有个大致的了解。

最后要说明的是，系统调用返回函数 ret_from_sys_call()是从系统调用、异常及中断返回函数通常要调用的函数，但并不是非得调用，对于那些要经常被响应的和要被尽快处理的中断请求信号，为了减少系统开销，处理完成后并不调用 ret_from_sys_call()（因为很明显，从这些中断处理程序返回到的用户空间肯定是那个被中断的进程，无需重新选择），并且，它们作的工作要尽可能少，因为响应的频率太高了。

Linux 调度程序和其他的 UNIX 调度程序不同，尤其是在“nice level”优先级的处理上，与优先权调度（priority 高的进程最先运行）不同，Linux 用的是时间片轮转调度（Round Robing），但同时又保证了高优先级的进程运行得既快、时间又长（both sooner and longer）。而标准的 UNIX 调度程序都用到了多级进程队列。大多数的实现都用到了二级优先队列：一个标准队列和一个实时（“real time”）队列。一般情况下，如果实时队列中的进程未被阻塞，它们都要在标准队列中的进程之前被执行，并且，每个队列中，“nice level”高的进程先被执行。

总体上，Linux 调度序程在交互性方面表现很出色，当然了，这是以牺牲一部分“吞吐量”为代价的。

5.3.3　进程调度的依据

调度程序运行时，要在所有处于可运行状态的进程之中选择最值得运行的进程投入运行。选择进程的依据是什么呢？在每个进程的 task_struct 结构中有如下 5 项：

need_resched、nice、counter、policy 及 rt_priority

（1）need_resched：在调度时机到来时，检测这个域的值，如果为 1，则调用 schedule()。

（2）counter：进程处于运行状态时所剩余的时钟滴答数，每次时钟中断到来时，这个值就减 1。当这个域的值变得越来越小，直至为 0 时，就把 need_resched 域置 1，因此，也

把这个域叫做进程的“动态优先级”。

（3）nice：进程的“静态优先级”，这个域决定 counter 的初值。只有通过 nice()、POSIX.1b sched_setparam() 或 5.4BSD/SVR4 setpriority()系统调用才能改变进程的静态优先级。

（4）rt_priority：实时进程的优先级

（5）policy：从整体上区分实时进程和普通进程，因为实时进程和普通进程的调度是不同的，它们两者之间，实时进程应该先于普通进程而运行，可以通过系统调用 sched_setscheduler（ ）来改变调度的策略。对于同一类型的不同进程，采用不同的标准来选择进程。对于普通进程，选择进程的主要依据为 counter 和 nice 。对于实时进程，Linux 采用了两种调度策略，即 FIFO（先来先服务调度）和 RR（时间片轮转调度）。因为实时进程具有一定程度的紧迫性，所以衡量一个实时进程是否应该运行，Linux 采用了一个比较固定的标准。实时进程的 counter 只是用来表示该进程的剩余滴答数，并不作为衡量它是否值得运行的标准，这和普通进程是有区别的。

这里再次说明，与其他操作系统一样，Linux 的时间单位也是“时钟滴答”，只是不同的操作系统对一个时钟滴答的定义不同而已（Linux 设计者将一个“时钟滴答”定义为 10ms）。在这里，我们把 counter 叫做进程的时间片，但实际上它仅仅是时钟滴答的个数，例如，若 counter 为 5，则分配给该进程的时间片就为 5 个时钟滴答，也就是 5*10ms=50ms，实际上，Linux 2.4 中给进程初始时间片的大小就是 50ms

5.3.4 进程可运行程度的衡量

函数 goodness()就是用来衡量一个处于可运行状态的进程值得运行的程度。该函数综合使用了上面我们提到的 5 项，给每个处于可运行状态的进程赋予一个权值（weight），调度程序以这个权值作为选择进程的唯一依据。函数主体如下（为了便于理解，笔者对函数做了一些改写和简化，只考虑单处理机的情况）：

```
static inline int goodness (struct task_struct * p, struct mm_struct *this_mm)
{    int weight;      ／＊ 权值，作为衡量进程是否运行的唯一依据 ＊

       weight=-1;
       if  (p->policy&SCHED_YIELD)
        goto out;  /*如果该进程愿意“礼让（yield）”，则让其权值为－1 ＊／
      switch (p->policy)
     {
         /* 实时进程*/
         case SCHED_FIFO:
         case SCHED_RR:
              weight = 1000 + p->rt_priority;

         /* 普通进程 */
         case SCHED_OTHER:
             {     weight = p->counter;
               if (!weight)
                 goto out
```

```
                /* 做细微的调整*/
                if  (p->mm=this_mm||!p->mm)
                        weight = weight+1;
                  weight+=20-p->nice;
                }
    }
out:
    return weight;   /*返回权值*/
}
```

其中，在 sched.h 中对调度策略定义如下：

```
#define SCHED_OTHER            0
#define SCHED_FIFO             1
#define SCHED_RR               2
    #define SCHED_YIELD             0x10
```

这个函数比较很简单。首先，根据 policy 区分实时进程和普通进程。实时进程的权值取决于其实时优先级，其至少是 1000，与 conter 和 nice 无关。普通进程的权值需特别说明如下两点。

(1) 为什么进行细微的调整？如果 p->mm 为空，则意味着该进程无用户空间（例如内核线程），则无需切换到用户空间。如果 p->mm=this_mm，则说明该进程的用户空间就是当前进程的用户空间，该进程完全有可能再次得到运行。对于以上两种情况，都给其权值加 1，算是对它们小小的“奖励”。

(2) 进程的优先级 nice 是从早期 UNIX 沿用下来的负向优先级，其数值标志“谦让”的程度，其值越大，就表示其越“谦让”，也就是优先级越低，其取值范围为－20～＋19，因此，（20-p->nice）的取值范围就是 0～40。可以看出，普通进程的权值不仅考虑了其剩余的时间片，还考虑了其优先级，优先级越高，其权值越大。

有了衡量进程是否应该运行的标准，选择进程就是轻而易举的事情了，“弱肉强食”，谁的权值大谁就先运行。

根据进程调度的依据，调度程序就可以控制系统中的所有处于可运行状态的进程并在它们之间进行选择。

5.3.5 进程调度的实现

调度程序在内核中就是一个函数，为了讨论方便，我们同样对其进行了简化，略去对 SMP 的实现部分。

```
asmlinkage void schedule (void)
{

  struct task_struct *prev, *next, *p; ／* prev 表示调度之前的进程,
     next 表示调度之后的进程 *／
    struct list_head *tmp;
    int this_cpu, c;

      if  (!current->active_mm)  BUG();/*如果当前进程的 active_mm 为空，出错*／
need_resched_back:
          prev = current;          ／*让 prev 成为当前进程 *／
```

```
            this_cpu = prev->processor;

    if  (in_interrupt())  {／*如果 schedule 是在中断服务程序内部执行，
    就说明发生了错误*／
            printk ("Scheduling in interrupt\n") ;
              BUG();
         }
      release_kernel_lock (prev, this_cpu) ; /*释放全局内核锁，
    并开 this_cpu 的中断*／
          spin_lock_irq (&runqueue_lock) ; ／*锁住运行队列，并且同时关中断*/
           if  (prev->policy == SCHED_RR)  ／*将一个时间片用完的 SCHED_RR 实时
                goto move_rr_last;       进程放到队列的末尾 *／
     move_rr_back:
           switch  (prev->state)  {      ／*根据 prev 的状态做相应的处理*／
                case TASK_INTERRUPTIBLE:  /*此状态表明该进程可以被信号中断*/
                       if  (signal_pending (prev) )  { /*如果该进程有未处理的
    信号，则让其变为可运行状态*/
                             prev->state = TASK_RUNNING;
                              break;
                       }
                 default:     ／*如果为可中断的等待状态或僵死状态*／
                       del_from_runqueue (prev) ; ／*从运行队列中删除*／
                 case TASK_RUNNING:;／*如果为可运行状态，继续处理*／
           }
           prev->need_resched = 0;

        ／*下面是调度程序的正文 *／
    repeat_schedule:    ／*真正开始选择值得运行的进程*／
           next = idle_task (this_cpu) ; /*缺省选择空闲进程*/
      c = -1000;
        if  (prev->state == TASK_RUNNING)
            goto still_running;
    still_running_back:
       list_for_each (tmp, &runqueue_head)  { ／*遍历运行队列*／
          p = list_entry (tmp, struct task_struct, run_list) ;
     if  ( can_schedule  ( p,  this_cpu )  )    {  ／ * 单  CPU  中 ，  该 函 数 总 返 回  1* ／
int weight = goodness (p, this_cpu, prev->active_mm) ;
                if  (weight > c)
                    c = weight, next = p;
            }
        }

    ／* 如果 c 为 0，说明运行队列中所有进程的权值都为 0，也就是分配给各个进程的
        时间片都已用完，需重新计算各个进程的时间片 */
       if  (!c)  {
               struct task_struct *p;
               spin_unlock_irq (&runqueue_lock) ;／*锁住运行队列*／
                read_lock (&tasklist_lock) ;  ／* 锁住进程的双向链表*／
               for_each_task (p)            ／* 对系统中的每个进程*／
               p->counter =  (p->counter >> 1)  + NICE_TO_TICKS (p->nice) ;
               read_unlock (&tasklist_lock) ;
                   spin_lock_irq (&runqueue_lock) ;
```

```
                goto repeat_schedule;
        }


    spin_unlock_irq (&runqueue_lock) ;／*对运行队列解锁，并开中断*／

      if  (prev == next)  {      /*如果选中的进程就是原来的进程*/
           prev->policy &= ~SCHED_YIELD;
              goto same_process;
     }

         ／* 下面开始进行进程切换*／
      kstat.context_swtch++; ／*统计上下文切换的次数*／

       {
              struct mm_struct *mm = next->mm;
              struct mm_struct *oldmm = prev->active_mm;
             if  (!mm)  {  ／*如果是内核线程，则借用 prev 的地址空间*／
                    if  (next->active_mm)  BUG();
                    next->active_mm = oldmm;

             } else { ／*如果是一般进程，则切换到 next 的用户空间*／
                     if  (next->active_mm != mm)  BUG();
                     switch_mm (oldmm, mm, next, this_cpu) ;
             }

           if  (!prev->mm)  { ／*如果切换出去的是内核线程*／
                prev->active_mm = NULL;／*归还它所借用的地址空间*／
                  mmdrop (oldmm) ;      ／*mm_struct 中的共享计数减 1*／
              }
       }

       switch_to (prev, next, prev) ; ／*进程的真正切换，即堆栈的切换*／
       __schedule_tail (prev) ;  ／*置 prev->policy 的 SCHED_YIELD 为 0 *／

same_process:
       reacquire_kernel_lock (current) ;／*针对 SMP*／
        if  (current->need_resched)      ／*如果调度标志被置位*／
              goto need_resched_back; ／*重新开始调度*／
        return;
}
```

以上就是调度程序的主要内容，为了对该程序形成一个清晰的思路，我们对其再给出进一步的解释。

- 如果当前进程既没有自己的地址空间,也没有向别的进程借用地址空间,那肯定出错。另外，如果 schedule()在中断服务程序内部执行,那也出错。

- 对当前进程做相关处理,为选择下一个进程做好准备。当前进程就是正在运行着的进程，可是，当进入 schedule()时，其状态却不一定是 TASK_RUNNIG，例如，在 exit()系统调用中，当前进程的状态可能已被改为 TASK_ZOMBE；又例如，在 wait4()系统调用中，当前进程的状态可能被置为 TASK_INTERRUPTIBLE。因此，如果当前进程处于这些状态中的一种，就

要把它从运行队列中删除。

- 从运行队列中选择最值得运行的进程，也就是权值最大的进程。
- 如果已经选择的进程其权值为 0，说明运行队列中所有进程的时间片都用完了（队列中肯定没有实时进程，因为其最小权值为 1000），因此，重新计算所有进程的时间片，其中宏操作 NICE_TO_TICKS 就是把优先级 nice 转换为时钟滴答。
- 进程地址空间的切换。如果新进程有自己的用户空间，也就是说，如果 next->mm 与 next->active_mm 相同，那么，switch_mm（ ）函数就把该进程从内核空间切换到用户空间，也就是加载 next 的页目录。如果新进程无用户空间（next->mm 为空），也就是说，如果它是一个内核线程，那它就要在内核空间运行，因此，需要借用前一个进程（prev）的地址空间，因为所有进程的内核空间都是共享的，因此，这种借用是有效的。
- 用宏 switch_to()进行真正的进程切换，后面将详细描述。

5.4 进程切换

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换，任务切换，或上下文切换。Intel 在 i386 系统结构的设计中考虑到了进程（任务）的管理和调度，并从硬件上支持任务之间的切换。

5.4.1 硬件支持

Intel i386 体系结构包括了一个特殊的段类型，叫任务状态段（TSS），如图 5.4 所示。每个任务包含有它自己最小长度为 104 字节的 TSS 段，在/include/ i386/processor.h 中定义为 tss_struct 结构：

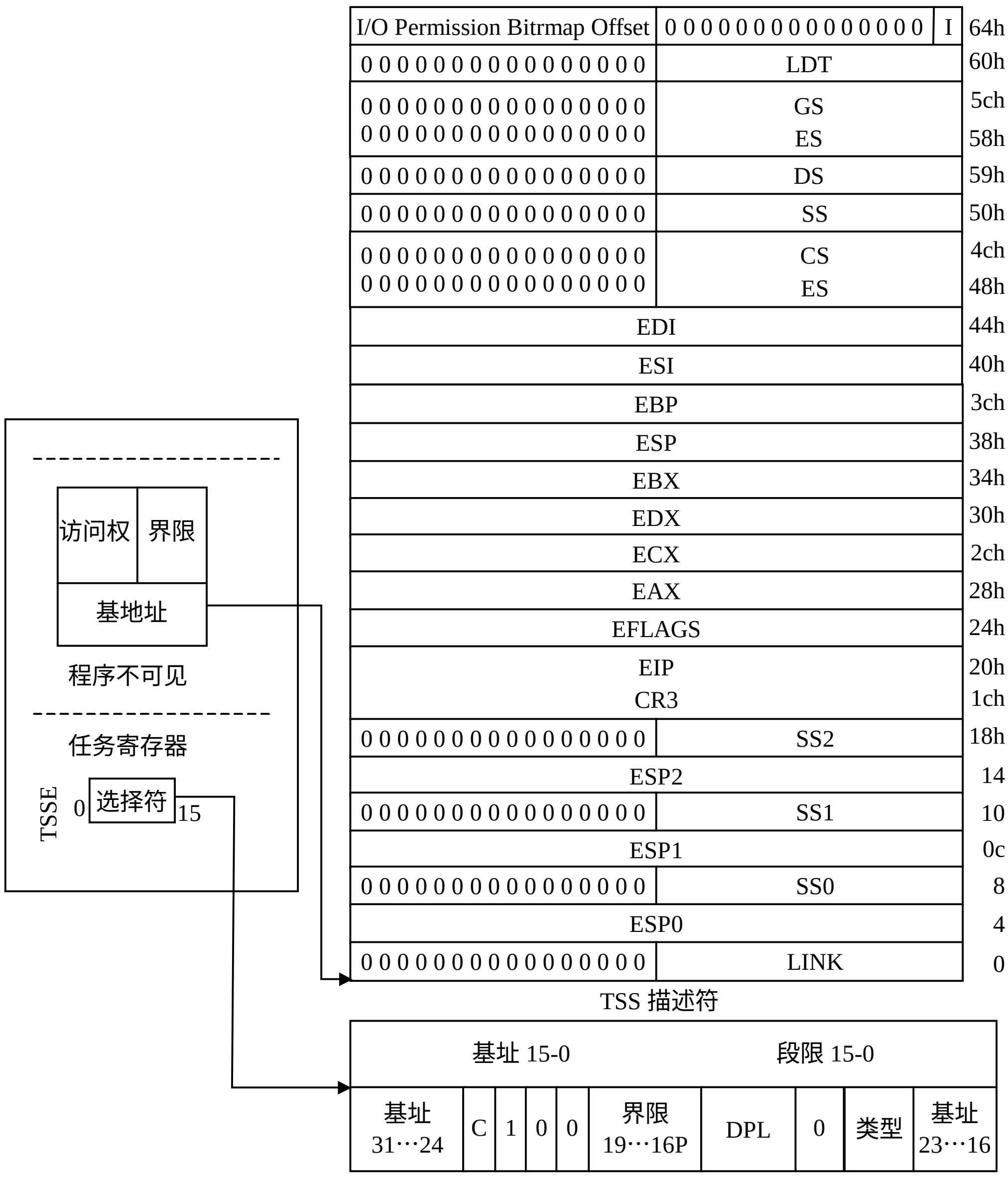


图 5.4　Intel i386 任务状态段及 TR 寄存器

```
struct tss_struct {
        unsigned short  back_link,__blh;
        unsigned long   esp0;
        unsigned short  ss0,__ss0h;／*0 级堆栈指针，即 Linux 中的内核级 *／
        unsigned long   esp1;
        unsigned short  ss1,__ss1h; ／* 1 级堆栈指针，未用*／
        unsigned long   esp2;
        unsigned short  ss2,__ss2h; ／* 2 级堆栈指针，未用*／
        unsigned long   __cr3;
        unsigned long   eip;
        unsigned long   eflags;
```

```
        unsigned long   eax,ecx,edx,ebx;
        unsigned long   esp;
        unsigned long   ebp;
        unsigned long   esi;
        unsigned long   edi;
        unsigned short  es, __esh;
        unsigned short  cs, __csh;
        unsigned short  ss, __ssh;
        unsigned short  ds, __dsh;
        unsigned short  fs, __fsh;
        unsigned short  gs, __gsh;
        unsigned short  ldt, __ldth;
        unsigned short  trace, bitmap;
        unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
         /*
        * pads the TSS to be cacheline-aligned  (size is 0x100)
         */
        unsigned long __cacheline_filler[5];
};
```

每个 TSS 有它自己 8 字节的任务段描述符（Task State Segment Descriptor ，简称 TSSD)。这个描述符包括指向 TSS 起始地址的 32 位基地址域，20 位界限域，界限域值不能小于十进制 104（由 TSS 段的最小长度决定)。TSS 描述符存放在 GDT 中，它是 GDT 中的一个表项。

后面将会看到，Linux 在进程切换时，只用到 TSS 中少量的信息，因此 Linux 内核定义了另外一个数据结构，这就是 thread_struct 结构：

```
struct thread_struct {
        unsigned long   esp0;
        unsigned long   eip;
        unsigned long   esp;
        unsigned long   fs;
        unsigned long   gs;
   /* Hardware debugging registers */
         unsigned long   debugreg[8];  /* %%db0-7 debug registers */
  /* fault info */
         unsigned long   cr2, trap_no, error_code;
   /* floating point info */
         union i387_union        i387;
  /* virtual 86 mode info */
        struct vm86_struct      * vm86_info;
         unsigned long           screen_bitmap;
        unsigned long           v86flags, v86mask, v86mode, saved_esp0;
/* IO permissions */
         int             ioperm;
        unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
};
```

用这个数据结构来保存 cr2 寄存器、浮点寄存器、调试寄存器及指定给 Intel 80x86 处理器的其他各种各样的信息。需要位图是因为 ioperm（ ） 及 iopl（ ）系统调用可以允许用户态的进程直接访问特殊的 I/O 端口。尤其是，如果把 eflag 寄存器中的 IOPL 域设置

为 3，就允许用户态的进程访问对应的 I/O 访问权位图位为 0 的任何一个 I/O 端口。

那么，进程到底是怎样进行切换的？

从第三章我们知道，在中断描述符表（IDT）中，除中断门、陷阱门和调用门外，还有一种“任务门”。任务门中包含有 TSS 段的选择符。当 CPU 因中断而穿过一个任务门时，就会将任务门中的段选择符自动装入 TR 寄存器，使 TR 指向新的 TSS，并完成任务切换。CPU 还可以通过 JMP 或 CALL 指令实现任务切换，当跳转或调用的目标段（代码段）实际上指向 GDT 表中的一个 TSS 描述符项时，就会引起一次任务切换。

Intel 的这种设计确实很周到，也为任务切换提供了一个非常简洁的机制。但是，由于 i386 的系统结构基本上是 CISC 的，通过 JMP 指令或 CALL（或中断）完成任务的过程实际上是“复杂指令”的执行过程，其执行过程长达 300 多个 CPU 周期（一个 POP 指令占 12 个 CPU 周期），因此，Linux 内核并不完全使用 i386 CPU 提供的任务切换机制。

由于 i386 CPU 要求软件设置 TR 及 TSS，Linux 内核只不过“走过场”地设置 TR 及 TSS，以满足 CPU 的要求。但是，内核并不使用任务门，也不使用 JMP 或 CALL 指令实施任务切换。内核只是在初始化阶段设置 TR，使之指向一个 TSS，从此以后再不改变 TR 的内容了。也就是说，每个 CPU（如果有多个 CPU）在初始化以后的全部运行过程中永远使用那个初始的 TSS。同时，内核也不完全依靠 TSS 保存每个进程切换时的寄存器副本，而是将这些寄存器副本保存在各个进程自己的内核栈中（参见上一章 task_struct 结构的存放）。

这样以来，TSS 中的绝大部分内容就失去了原来的意义。那么，当进行任务切换时，怎样自动更换堆栈？我们知道，新任务的内核栈指针（SS0 和 ESP0）应当取自当前任务的 TSS，可是，Linux 中并不是每个任务就有一个 TSS，而是每个 CPU 只有一个 TSS。Intel 原来的意图是让 TR 的内容（即 TSS）随着任务的切换而走马灯似地换，而在 Linux 内核中却成了只更换 TSS 中的 SS0 和 ESP0，而不更换 TSS 本身，也就是根本不更换 TR 的内容。这是因为，改变 TSS 中 SS0 和 ESP0 所化的开销比通过装入 TR 以更换一个 TSS 要小得多。因此，在 Linux 内核中，TSS 并不是属于某个进程的资源，而是全局性的公共资源。在多处理机的情况下，尽管内核中确实有多个 TSS，但是每个 CPU 仍旧只有一个 TSS。

5.4.2　进程切换

前面所介绍的 schedule()中调用了 switch_to 宏，这个宏实现了进程之间的真正切换，其代码存放于 include/ i386/system.h：

```
#define switch_to (prev,next,last)  do {                              \
      asm volatile ("pushl %%esi\n\t"                               \
                   "pushl %%edi\n\t"                                \
                   "pushl %%ebp\n\t"                                \
                   "movl %%esp,%0\n\t"        /* save ESP */        \
                   "movl %3,%%esp\n\t"        /* restore ESP */     \
                   "movl $1f,%1\n\t"          /* save EIP */        \
                   "pushl %4\n\t"             /* restore EIP */     \
                   "jmp __switch_to\n"                              \
                    "1:\t"                                          \
                   "popl %%ebp\n\t"                                 \
                   "popl %%edi\n\t"                                 \
```

```
13                      "popl %%esi\n\t"                                  \
14                      :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
15                       "=b" (last)                                       \
16                      :"m" (next->thread.esp),"m" (next->thread.eip),    \
17                       "a" (prev), "d" (next),                           \
18                       "b" (prev));                                      \
19 } while (0)
```

switch_to 宏是用嵌入式汇编写成，比较难理解，为描述方便起见，我们给代码编了行号，在此我们给出具体的解释。

- thread 的类型为前面介绍的 thread_struct 结构。
- 输出参数有 3 个，表示这段代码执行后有 3 项数据会有变化，它们与变量及寄存器的对应关系如下：

0%与 prev->thread.esp 对应，1%与 prev->thread.eip 对应，这两个参数都存放在内存，而 2%与 ebx 寄存器对应，同时说明 last 参数存放在 ebx 寄存器中。

- 输入参数有 5 个，其对应关系如下：

3%与 next->thread.esp 对应，4%与 next->thread.eip 对应，这两个参数都存放在内存，而 5%、6%和 7%分别与 eax、edx 及 ebx 相对应，同时说明 prev、next 以及 prev 这 3 个参数分别放在这 3 个寄存器中。表 5.1 列出了这几种对应关系。

- 第 2～4 行就是在当前进程 prev 的内核栈中保存 esi、edi 及 ebp 寄存器的内容。
- 第 5 行将 prev 的内核堆栈指针 ebp 存入 prev->thread.esp 中。
- 第 6 行把将要运行进程 next 的内核栈指针 next->thread.esp 置入 esp 寄存器中。从现在开始，内核对 next 的内核栈进行操作，因此，这条指令执行从 prev 到 next 真正的上下文切换，因为进程描述符的地址与其内核栈的地址紧紧地联系在一起（参见第四章），因此，改变内核栈就意味着改变当前进程。如果此处引用 current，那就已经指向 next 的 task_struct 结构了。从这个意义上说，进程的切换在这一行指令执行完以后就已经完成。但是，构成一个进程的另一个要素是程序的执行，这方面的切换尚未完成。

表 5.1　　　　输入/输出参数与变量及寄存器的对应关系

参数类型	参数名	内存变量	寄存器	函数参数
输出参数	0%	prev->thread.esp		
	1%	prev->thread.eip		
	2%		ebx	last
输入参数	3%	next->thread.esp		
	4%	next->thread.eip		
	5%		eax	prev
	6%		edx	next
	7%		ebx	prev

- 第 7 行将标号“1”所在的地址，也就是第一条 popl 指令（第 11 行）所在的地址保

存在 prev->thread.eip 中，这个地址就是 prev 下一次被调度运行而切入时的“ 返回 ”地址。

- 第 8 行将 next->thread.eip 压入 next 的内核栈。那么，next->thread.eip 究竟指向那个地址？实际上，它就是 next 上一次被调离时通过第 7 行保存的地址，也就是第 11 行 popl 指令的地址。因为，每个进程被调离时都要执行这里的第 7 行，这就决定了每个进程（除了新创建的进程）在受到调度而恢复执行时都从这里的第 11 行开始。
- 第 9 行通过 jump 指令（而不是 call 指令）转入一个函数__switch_to()。这个函数的具体实现将在下面介绍。当 CPU 执行到__switch_to()函数的 ret 指令时，最后进入堆栈的 next->thread.eip 就变成了返回地址，这就是标号“ 1 ”的地址。
- 第 11～13 行恢复 next 上次被调离时推进堆栈的内容。从现在开始，next 进程就成为当前进程而真正开始执行。

下面我们来讨论__switch_to()函数。

在调用__switch_to()函数之前，对其定义了 fastcall ：

```
extern void FASTCALL (__switch_to (struct task_struct *prev, struct task_struct *next) ) ;
```

fastcall 对函数的调用不同于一般函数的调用，因为__switch_to()从寄存器(如表 5.1)取参数，而不像一般函数那样从堆栈取参数，也就是说，通过寄存器 eax 和 edx 把 prev 和 next 参数传递给__switch_to()函数。

```
void __switch_to (struct task_struct *prev_p, struct task_struct *next_p)
{
        struct thread_struct *prev = &prev_p->thread,
                                *next = &next_p->thread;
        struct tss_struct *tss = init_tss + smp_processor_id();

      unlazy_fpu (prev_p) ;／* 如果数学处理器工作，则保存其寄存器的值*／

  /* 将 TSS 中的内核级（0 级）堆栈指针换成 next->esp0，这就是 next 进程在内核
     栈的指针*/

      tss->esp0 = next->esp0;

    ／＊ 保存 fs 和 gs，但无需保存 es 和 ds，因为当处于内核时，内核段
总是保持不变＊／

       asm volatile ("movl %%fs,%0":"=m"  (* (int *) &prev->fs) ) ;
       asm volatile ("movl %%gs,%0":"=m"  (* (int *) &prev->gs) ) ;

     ／＊恢复 next 进程的 fs 和 gs ＊／

         loadsegment (fs, next->fs) ;
        loadsegment (gs, next->gs) ;

／＊ 如果 next 挂起时使用了调试寄存器，则装载 0～7 个寄存器中的 6 个寄存器，其中第 4、5 个寄存器没有使用 ＊／

        if  (next->debugreg[7]) {
                loaddebug (next, 0) ;
                loaddebug (next, 1) ;
```

```
                loaddebug (next, 2) ;
                loaddebug (next, 3) ;
                 /* no 4 and 5 */
                loaddebug (next, 6) ;
                loaddebug (next, 7) ;
        }

         if  (prev->ioperm || next->ioperm)  {
                if  (next->ioperm)  {

／＊把 next 进程的 I/O 操作权限位图拷贝到 TSS 中 ＊／
                       memcpy (tss->io_bitmap, next->io_bitmap,
IO_BITMAP_SIZE*sizeof (unsigned long) ) ;

/* 把 io_bitmap 在 tss 中的偏移量赋给 tss->bitmap ＊／
                         tss->bitmap = IO_BITMAP_OFFSET;
                 } else

／＊如果一个进程要使用 I/O 指令，但是，若位图的偏移量超出 TSS 的范围，
就会产生一个可控制的 SIGSEGV 信号。第一次对 sys_ioperm()的调用会
建立起适当的位图  ＊／

                        tss->bitmap = INVALID_IO_BITMAP_OFFSET;
         }
}
```

从上面的描述我们看到，尽管 Intel 本身为操作系统中的进程（任务）切换提供了硬件支持，但是 Linux 内核的设计者并没有完全采用这种思想，而是用软件实现了进程切换，而且，软件实现比硬件实现的效率更高，灵活性更大。

第六章　Linux 内存管理

存储器是一种必须仔细管理的重要资源。在理想的情况下，每个程序员都喜欢无穷大、快速并且内容不易变（即掉电后内容不会丢失）的存储器，同时又希望它是廉价的。但不幸的是，当前技术没有能够提供这样的存储器，因此大部分的计算机都有一个存储器层次结构，即少量、快速、昂贵、易变的高速缓存（cache）；若干兆字节的中等速度、中等价格、易变的主存储器（RAM）；数百兆或数千兆的低速、廉价、不易变的磁盘。如图 6.1 所示，这些资源的合理使用与否 ，直接关系着系统的效率。

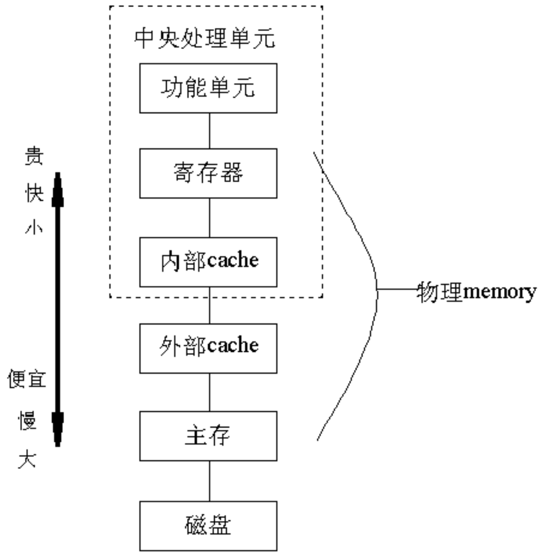


图 6.1　存储器的层次结构

Linux 内存管理是内核最复杂的任务之一。主要是因为它用到许多 CPU 提供的功能，而且和这些功能密切相关。正因为如此，我们在第二章较详细地介绍了 Intel 386 的段机制和页机制，可以说这是进行内存管理分析的物质基础 。这一章用大量的篇幅描述了 Linux 内存管理所涉及到的各种机制，如内存的初始化机制、地址映射机制、请页机制、交换机制、内存分配和回收机制、缓存和刷新机制以及内存共享机制，最后还分析了程序的创建和执行。

6.1　Linux 的内存管理概述

Linux 是为多用户多任务设计的操作系统，所以存储资源要被多个进程有效共享；且由

于程序规模的不断膨胀，要求的内存空间比从前大得多。 Linux 内存管理的设计充分利用了计算机系统所提供的虚拟存储技术，真正实现了虚拟存储器管理。

第二章介绍的 Intel 386 的段机制和页机制是 Linux 实现虚拟存储管理的一种硬件平台。实际上， Linux 2.0 以上的版本不仅仅可以运行在 Intel 系列个人计算机上，还可以运行在 Apple 系列、DEC Alpha 系列、MIPS 和 Motorola 68k 等系列上， 这些平台都支持虚拟存储器管理，我们之所以选择 Intel 386，是因为它具有代表性和普遍性。

Linux 的内存管理主要体现在对虚拟内存的管理。我们可以把 Linux 虚拟内存管理功能概括为以下几点：

- 大地址空间；
- 进程保护；
- 内存映射；
- 公平的物理内存分配；
- 共享虚拟内存。

关于这些功能的实现，我们将会陆续介绍。

6.1.1 Linux 虚拟内存的实现结构

我们先从整体结构上了解 Linux 对虚拟内存的实现结构，如图 6.2 所示。

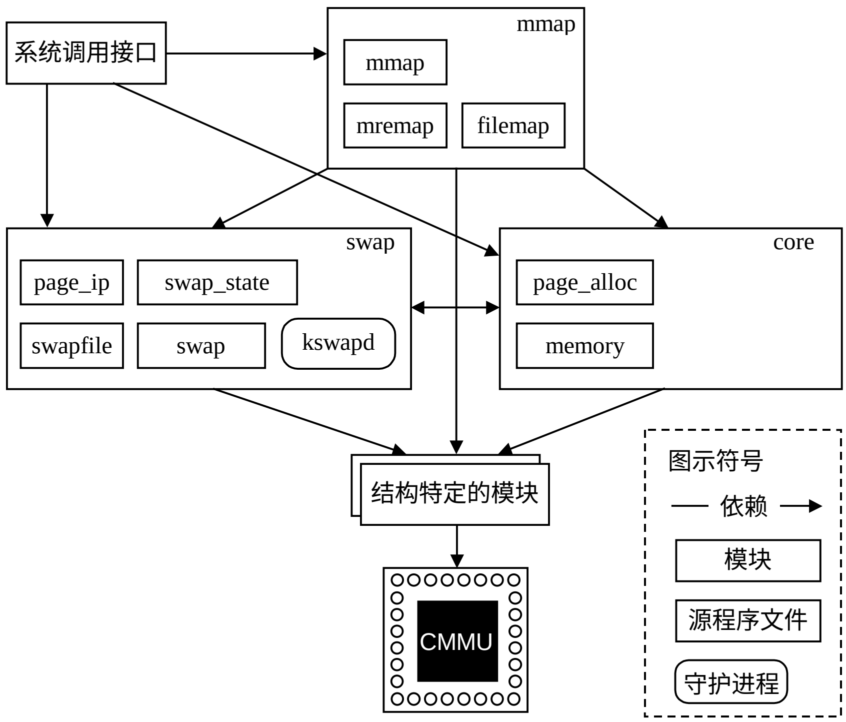


图 6.2 Linux 虚拟内存的实现结构

从图 6.2 中可看到实现虚拟内存的组成模块。其实现的源代码大部分放在/mm 目录下。

（1）内存映射模块(mmap)：负责把磁盘文件的逻辑地址映射到虚拟地址，以及把虚拟地址映射到物理地址。

（2）交换模块（swap）：负责控制内存内容的换入和换出，它通过交换机制，使得在物理内存的页面（RAM 页）中保留有效的页 ，即从主存中淘汰最近没被访问的页，保存近来访问过的页。

（3）核心内存管理模块（core）：负责核心内存管理功能，即对页的分配、回收、释放及请页处理等，这些功能将被别的内核子系统（如文件系统）使用。

（4）结构特定的模块：负责给各种硬件平台提供通用接口，这个模块通过执行命令来改变硬件 MMU 的虚拟地址映射，并在发生页错误时，提供了公用的方法来通知别的内核子系统。这个模块是实现虚拟内存的物理基础。

6.1.2　内核空间和用户空间

从第二章我们知道，Linux 简化了分段机制，使得虚拟地址与线性地址总是一致，因此，Linux 的虚拟地址空间也为 0～4G 字节。Linux 内核将这 4G 字节的空间分为两部分。将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为“内核空间”。而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有 4G 字节的虚拟空间。图 6.3 给出了进程虚拟空间示意图。

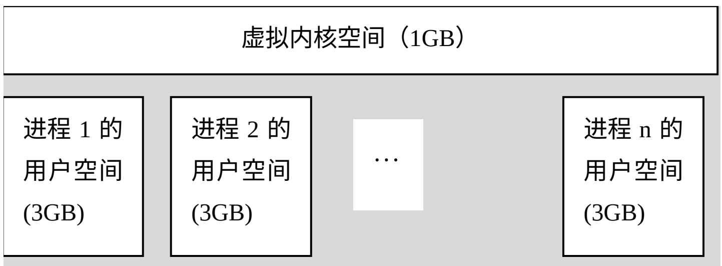


图 6.3　Linux 进程的虚拟空间

Linux 使用两级保护机制：0 级供内核使用，3 级供用户程序使用。从图 6.3 中可以看出，每个进程有各自的私有用户空间（0～3G），这个空间对系统中的其他进程是不可见的。最高的 1G 字节虚拟内核空间则为所有进程以及内核所共享。

1．虚拟内核空间到物理空间的映射

内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。读者会问，系统启动时，内核的代码和数据不是被装入到物理内存吗？它们为什么也处于虚拟内存中呢？这和编译程序有关，后面我们通过具体讨论就会明白这一点。

虽然内核空间占据了每个虚拟空间中的最高 1G 字节，但映射到物理内存却总是从最低

地址（0x00000000）开始。如图 6.4 所示，对内核空间来说，其地址映射是很简单的线性映射，0xC0000000 就是物理地址与线性地址之间的位移量，在 Linux 代码中就叫做 PAGE_OFFSET。

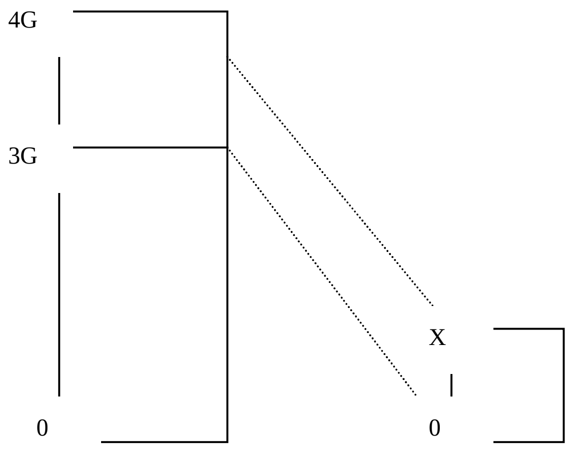


图 6.4　内核的虚拟地址空间到物理地址空间的映射

我们来看一下在 include/asm/i386/page.h 中对内核空间中地址映射的说明及定义：

```
/*
* This handles the memory map.. We could make this a config
* option, but too many people screw it up, and too few need
* it.
*
* A __PAGE_OFFSET of 0xC0000000 means that the kernel has
* a virtual address space of one gigabyte, which limits the
* amount of physical memory you can use to about 950MB.
*
* If you want more physical memory than this then see the CONFIG_HIGHMEM4G
* and CONFIG_HIGHMEM64G options in the kernel configuration.
*/

#define __PAGE_OFFSET            (0xC0000000)
……
#define PAGE_OFFSET              ((unsigned long)__PAGE_OFFSET)
#define __pa(x)                   ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)                   ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

源代码的注释中说明，如果你的物理内存大于 950MB，那么在编译内核时就需要加 CONFIG_HIGHMEM4G 和 CONFIG_HIGHMEM64G 选项，这种情况我们暂不考虑。如果物理内存小于 950MB，则对于内核空间而言，给定一个虚地址 x，其物理地址为“x- PAGE_OFFSET”，给定一个物理地址 x，其虚地址为“x+ PAGE_OFFSET”。

这里再次说明，宏__pa()仅仅把一个内核空间的虚地址映射到物理地址，而决不适用于用户空间，用户空间的地址映射要复杂得多。

2．内核映像

在下面的描述中，我们把内核的代码和数据就叫内核映像（Kernel Image)。当系统启动时，Linux 内核映像被安装在物理地址 0x00100000 开始的地方，即 1MB 开始的区间(第 1M 留作它用)。然而，在正常运行时， 整个内核映像应该在虚拟内核空间中，因此，连接程序在连接内核映像时，在所有的符号地址上加一个偏移量 PAGE_OFFSET，这样，内核映像在内核空间的起始地址就为 0xC0100000。

例如，进程的页目录 PGD（属于内核数据结构）就处于内核空间中。在进程切换时，要将寄存器 CR3 设置成指向新进程的页目录 PGD，而该目录的起始地址在内核空间中是虚地址，但 CR3 所需要的是物理地址，这时候就要用__pa()进行地址转换。在 mm_context.h 中就有这么一行语句：

```
asm volatile（“movl %0,%%cr3”：：”r” （__pa（next->pgd））；
```

这是一行嵌入式汇编代码，其含义是将下一个进程的页目录起始地址 next_pgd，通过__pa()转换成物理地址，存放在某个寄存器中，然后用 mov 指令将其写入 CR3 寄存器中。经过这行语句的处理，CR3 就指向新进程 next 的页目录表 PGD 了。

6.1.3 虚拟内存实现机制间的关系

Linux 虚拟内存的实现需要各种机制的支持，因此，本章我们将对内存的初始化进行描述以后，围绕以下几种实现机制进行介绍：

- 内存分配和回收机制；
- 地址映射机制；
- 缓存和刷新机制；
- 请页机制；
- 交换机制；
- 内存共享机制。

这几种机制的关系如图 6.5 所示。

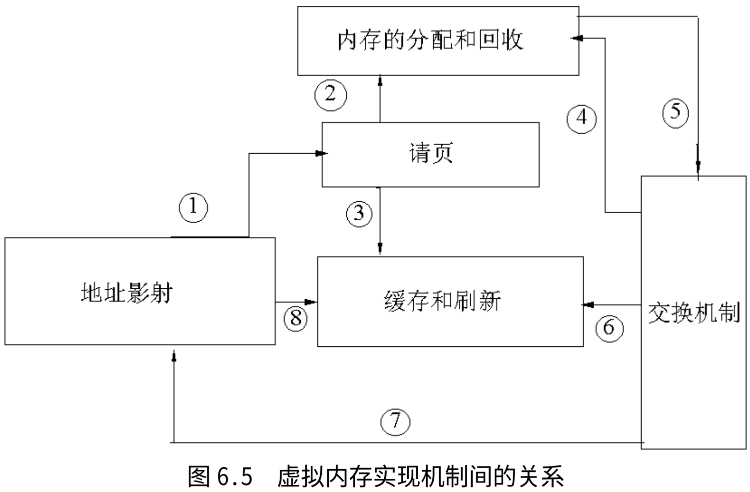


图 6.5 虚拟内存实现机制间的关系

首先内存管理程序通过映射机制把用户程序的逻辑地址映射到物理地址，在用户程序运行时如果发现程序中要用的虚地址没有对应的物理内存时，就发出了请页要求①；如果有空闲的内存可供分配，就请求分配内存②（于是用到了内存的分配和回收），并把正在使用的物理页记录在页缓存中③（使用了缓存机制）。如果没有足够的内存可供分配，那么就调用交换机制，腾出一部分内存④⑤。另外在地址映射中要通过 TLB（翻译后援存储器）来寻找物理页⑧；交换机制中也要用到交换缓存⑥，并且把物理页内容交换到交换文件中后也要修改页表来映射文件地址⑦。

6.2 Linux 内存管理的初始化

在对内存管理（MM）的各种机制介绍之前，首先需要对 MM 的初始化过程有所了解，以下介绍的内容是以第二章分段和分页机制为基础的，因此，读者应该先温习一下相关内容。因为在嵌入式操作系统的开发中，内存的初始化是重点关注的内容之一，因此本节将对内存的初始化给予详细描述。

6.2.1 启用分页机制

当 Linux 启动时，首先运行在实模式下，随后就要转到保护模式下运行。因为在第二章段机制中，我们已经介绍了 Linux 对段的设置，在此我们主要讨论与分页机制相关的问题。Linux 内核代码的入口点就是/arch/i386/kernel/head.S 中的 startup_32。

1．页表的初步初始化

```
/*
  * The page tables are initialized to only 8MB here - the final page
  * tables are set up later depending on memory size.
  */
.org 0x2000
ENTRY (pg0)

.org 0x3000
ENTRY (pg1)

/*
  * empty_zero_page must immediately follow the page tables ! (The
  * initialization loop counts until empty_zero_page)
  */

.org 0x4000
ENTRY (empty_zero_page)

 /*
  * Initialize page tables
  */
```

```
          movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
          movl $007,%eax          /* "007" doesn't mean with right to kill, but
                                    PRESENT+RW+USER */
 2:       stosl
          add $0x1000,%eax
          cmp $empty_zero_page-__PAGE_OFFSET,%edi
          jne 2b
```

内核的这段代码执行时，因为页机制还没有启用，还没有进入保护模式，因此指令寄存器 EIP 中的地址还是物理地址，但因为 pg0 中存放的是虚拟地址（gcc 编译内核以后形成的符号地址都是虚拟地址），因此，“ $pg0-__PAGE_OFFSET ”获得 pg0 的物理地址，可见 pg0 存放在相对于内核代码起点为 0x2000 的地方,即物理地址为 0x00102000，而 pg1 的物理地址则为 0x00103000。Pg0 和 pg1 这个两个页表中的表项则依次被设置为 0x007、0x1007、0x2007 等。其中最低的 3 位均为 1，表示这两个页为用户页，可写，且页的内容在内存中（参见图 2.24)。所映射的物理页的基地址则为 0x0、0x1000、0x2000 等，也就是物理内存中的页面 0、1、2、3 等等，共映射 2K 个页面，即 8MB 的存储空间。由此可以看出，Linux 内核对物理内存的最低要求为 8MB。紧接着存放的是 empty_zero_page 页(即零页)，零页存放的是系统启动参数和命令行参数，具体内容参见第十三章。

2．启用分页机制

```
/*
  * This is initialized to create an identity-mapping at 0-8M  (for bootup
  * purposes)  and another mapping of the 0-8M area at virtual address
  * PAGE_OFFSET.
  */
.org 0x1000
ENTRY (swapper_pg_dir)
          .long 0x00102007
          .long 0x00103007
          .fill BOOT_USER_PGD_PTRS-2,4,0
          /* default: 766 entries */
          .long 0x00102007
          .long 0x00103007
          /* default: 254 entries */
         .fill BOOT_KERNEL_PGD_PTRS-2,4,0
/*

 * Enable paging
 */
3:
     movl $swapper_pg_dir-__PAGE_OFFSET,%eax
     movl %eax,%cr3     /* set the page table pointer.. */
     movl %cr0,%eax
     orl $0x80000000,%eax
     movl %eax,%cr0     /* ..and set paging  (PG)  bit */
     jmp 1f             /* flush the prefetch-queue */
1:
     movl $1f,%eax
```

```
        jmp *%eax      /* make sure eip is relocated */
1:
```

我们先来看这段代码的功能。这段代码就是把页目录 swapper_pg_dir 的物理地址装入控制寄存器 cr3，并把 cr0 中的最高位置成 1，这就开启了分页机制。

但是，启用了分页机制，并不说明 Linux 内核真正进入了保护模式，因为此时，指令寄存器 EIP 中的地址还是物理地址，而不是虚地址。“ jmp 1f ”指令从逻辑上说不起什么作用，但是，从功能上说它起到丢弃指令流水线中内容的作用（这是 Intel 在 i386 技术资料中所建议的），因为这是一个短跳转，EIP 中还是物理地址。紧接着的 mov 和 jmp 指令把第 2 个标号为 1 的地址装入 EAX 寄存器并跳转到那儿。在这两条指令执行的过程中，EIP 还是指向物理地址“ 1MB＋某处 ”。因为编译程序使所有的符号地址都在虚拟内存空间中，因此，第 2 个标号 1 的地址就在虚拟内存空间的某处（PAGE_OFFSET+某处），于是，jmp 指令执行以后，EIP 就指向虚拟内核空间的某个地址，这就使 CPU 转入了内核空间，从而完成了从实模式到保护模式的平稳过渡。

然后再看页目录 swapper_pg_dir 中的内容。从前面的讨论我们知道 pg0 和 pg1 这两个页表的起始物理地址分别为 0x00102000 和 0x00103000。从图 2.22 可知，页目录项的最低 12 位用来描述页表的属性。因此，在 swapper_pg_dir 中的第 0 和第 1 个目录项 0x00102007、0x00103007，就表示 pg0 和 pg1 这两个页表是用户页表、可写且页表的内容在内存。

接着，把 swapper_pg_dir 中的第 2～767 共 766 个目录项全部置为 0。因为一个页表的大小为 4KB，每个表项占 4 个字节，即每个页表含有 1024 个表项，每个页的大小也为 4KB，因此这 768 个目录项所映射的虚拟空间为 768×1024×4K＝3G，也就是 swapper_pg_dir 表中的前 768 个目录项映射的是用户空间。

最后，在第 768 和 769 个目录项中又存放 pg0 和 pg1 这两个页表的地址和属性，而把第 770～1023 共 254 个目录项置 0。这 256 个目录项所映射的虚拟地址空间为 256×1024×4K＝1G，也就是 swapper_pg_dir 表中的后 256 个目录项映射的是内核空间。

由此可以看出，在初始的页目录 swapper_pg_dir 中，用户空间和内核空间都只映射了开头的两个目录项，即 8MB 的空间，而且有着相同的映射，如图 6.6 所示。

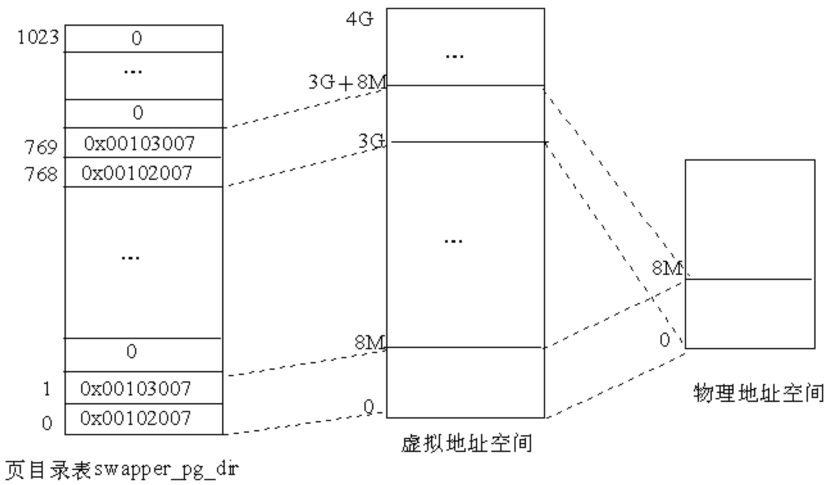


图 6.6　初始页目录 swapper_pg_dir 的映射图

读者会问，内核开始运行后运行在内核空间，那么，为什么把用户空间的低区（8M）也进行映射，而且与内核空间低区的映射相同？简而言之，是为了从实模式到保护模式的平稳过渡。具体地说，当 CPU 进入内核代码的起点 startup_32 后，是以物理地址来取指令的。在这种情况下，如果页目录只映射内核空间，而不映射用户空间的低区，则一旦开启页映射机制以后就不能继续执行了，这是因为，此时 CPU 中的指令寄存器 EIP 仍指向低区，仍会以物理地址取指令，直到以某个符号地址为目标作绝对转移或调用子程序为止。所以，Linux 内核就采取了上述的解决办法。

但是，在 CPU 转入内核空间以后，应该把用户空间低区的映射清除掉。后面读者将会看到，页目录 swapper_pg_dir 经扩充后就成为所有内核线程的页目录。在内核线程的正常运行中，处于内核态的 CPU 是不应该通过用户空间的虚拟地址访问内存的。清除了低区的映射以后，如果发生 CPU 在内核中通过用户空间的虚拟地址访问内存，就可以因为产生页面异常而捕获这个错误。

3．物理内存的初始分布

经过这个阶段的初始化，初始化阶段页目录及几个页表在物理空间中的位置如图 6.7 所示。

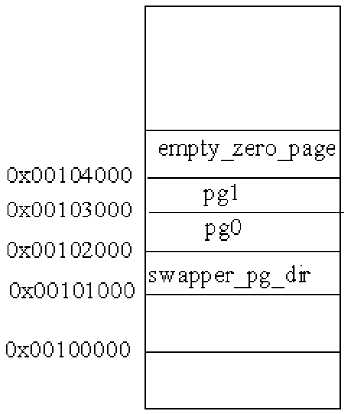


图 6.7　初始化阶段页目录及几个页表在物理空间中的位置

其中 empty_zero_page 中存放的是在操作系统的引导过程中所收集的一些数据，叫做引导参数。因为这个页面开始的内容全为 0，所以叫做“零页”，代码中常常通过宏定义 ZERO_PAGE 来引用这个页面。不过，这个页面要到初始化完成，系统转入正常运行时才会用到。为了后面内容介绍的方便，我们看一下复制到这个页面中的命令行参数和引导参数。这里假定这些参数已被复制到“零页”，在 setup.c 中定义了引用这些参数的宏：

```
 /*
  * This is set up by the setup-routine at boot-time
  */
#define PARAM    ( (unsigned char *) empty_zero_page)
#define SCREEN_INFO  (* (struct screen_info *)  (PARAM+0) )
```

```
#define EXT_MEM_K  (* (unsigned short *)   (PARAM+2) )
#define ALT_MEM_K  (* (unsigned long *)   (PARAM+0x1e0) )
#define E820_MAP_NR  (* (char*)   (PARAM+E820NR) )
#define E820_MAP     ( (struct e820entry *)   (PARAM+E820MAP) )
#define APM_BIOS_INFO  (* (struct apm_bios_info *)   (PARAM+0x40) )
#define DRIVE_INFO  (* (struct drive_info_struct *)   (PARAM+0x80) )
#define SYS_DESC_TABLE  (* (struct sys_desc_table_struct*)  (PARAM+0xa0) )
#define MOUNT_ROOT_RDONLY  (* (unsigned short *)   (PARAM+0x1F2) )
#define RAMDISK_FLAGS  (* (unsigned short *)   (PARAM+0x1F8) )
#define ORIG_ROOT_DEV  (* (unsigned short *)   (PARAM+0x1FC) )
#define AUX_DEVICE_INFO  (* (unsigned char *)   (PARAM+0x1FF) )
#define LOADER_TYPE  (* (unsigned char *)   (PARAM+0x210) )
#define KERNEL_START  (* (unsigned long *)   (PARAM+0x214) )
#define INITRD_START  (* (unsigned long *)   (PARAM+0x218) )
#define INITRD_SIZE  (* (unsigned long *)   (PARAM+0x21c) )
#define COMMAND_LINE  ( (char *)   (PARAM+2048) )
#define COMMAND_LINE_SIZE 256
```

其中宏 PARAM 就是 empty_zero_page 的起始位置，随着代码的阅读，读者会逐渐理解这些参数的用途。这里要特别对宏 E820_MAP 进行说明。E820_MAP 是个 struct e820entry 数据结构的指针，存放在参数块中位移为 0x2d0 的地方。这个数据结构定义在 include/i386/e820.h 中：

```
struct e820map {
     int nr_map;
     struct e820entry {
         unsigned long long addr;        /* start of memory segment */
         unsigned long long size;        /* size of memory segment */
         unsigned long type;             /* type of memory segment */
     } map[E820MAX];
};

extern struct e820map e820;
```

其中，E820MAX 被定义为 32。从这个数据结构的定义可以看出，每个 e820entry 都是对一个物理区间的描述，并且一个物理区间必须是同一类型。如果有一片地址连续的物理内存空间，其一部分是 RAM，而另一部分是 ROM，那就要分成两个区间。即使同属 RAM，如果其中一部分要保留用于特殊目的,那也属于不同的分区。在 e820.h 文件中定义了 4 种不同的类型：

```
#define E820_RAM        1
#define E820_RESERVED   2
#define E820_ACPI       3 /* usable as RAM once ACPI tables have been read */
#define E820_NVS        4

#define HIGH_MEMORY      (1024*1024)
```

其中 E820_NVS 表示“Non-Volatile Storage”，即“不挥发”存储器，包括 ROM、EPROM、Flash 存储器等。

在 PC 中，对于最初 1MB 存储空间的使用是特殊的。开头 640KB（0x0~0x9FFFF）为 RAM，从 0xA0000 开始的空间则用于 CGA、EGA、VGA 等图形卡。现在已经很少使用这些图形卡，但是不管是什么图形卡，开机时总是工作于 EGA 或 VGA 模式。从 0xF0000 开始到 0xFFFFF，即最高的 4KB，就是在 EPROM 或 Flash 存储器中的 BIOS。所以，只要有 BIOS 存在，就至少有两

个区间，如果 nr_map 小于 2，那就一定出错了。由于 BIOS 的存在，本来连续的 RAM 空间就不连续了。当然，现在已经不存在这样的存储结构了。1MB 的边界早已被突破，但因为历史的原因，把 1MB 以上的空间定义为“ HIGH_MEMORY ”，这个称呼一直沿用到现在，于是代码中的常数 HIGH_MEMORY 就定义为“ 1024×1024 ”。现在，配备了 128MB 的内存已经是很普遍了。但是，为了保持兼容，就得留出最初 1MB 的空间。

这个阶段初始化后，物理内存中内核映像的分布如图 6.8 所示。

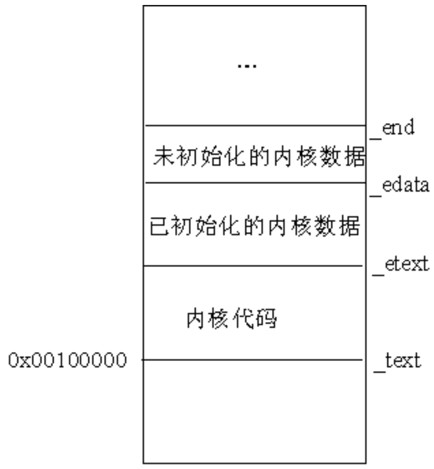


图 6.8　内核映像在物理内存中的分布

符号_text 对应物理地址 0x00100000，表示内核代码的第一个字节的地址。内核代码的结束位置用另一个类似的符号_etext 表示。内核数据被分为两组：初始化过的数据和未初始化过的数据。初始化过的数据在_etext 后开始，在_edata 处结束，紧接着是未初始化过的数据，其结束符号为_end，这也是整个内核映像的结束符号。

图中出现的符号是由编译程序在编译内核时产生的。你可以在 System.map 文件中找到这些符号的线性地址（或叫虚拟地址），System.map 是编译内核以后所创建的。

6.2.2　物理内存的探测

我们知道，BIOS 不仅能引导操作系统，还担负着加电自检和对资源的扫描探测，其中就包括了对物理内存的自检和扫描（你刚开机时所看到的信息就是此阶段 BIOS 显示的信息）。对于这个阶段中获得的内存信息可以通过 BIOS 调用“ int 0x15 ”加以检查。由于 Linux 内核不能作 BIOS 调用，因此内核本身就得代为检查，并根据获得的信息生成一幅物理内存构成图，这就是上面所介绍的 e820 图，然后通过上面提到的参数块传给内核。使得内核能知道系统中内存资源的配置。之所以称为 e820 图，是因为在通过 ” int 0x15 ” 查询内存的构成时要把调用参数之一设置成 0xe820。

分页机制启用以后，与内存管理相关的操作就是调用 init/main.c 中的 start_kernel（）函数，start_kernel（）函数要调用一个叫 setup_arch()的函数，setup_arch（）位于 arch/i386/kernel/setup.c 文件中，我们所关注的与物理内存探测相关的内容就在这个函数

中。

1．setup_arch()函数

这个函数比较繁琐和冗长，下面我们只对 setup_arch () 中与内存相关的内容给予描述。

首先调用 setup_memory_region()函数，这个函数处理内存构成图（map），并把内存的分布信息存放在全局变量 e820 中，后面会对此函数进行具体描述。

调用 parse_mem_cmdline(cmdline_p)函数。在特殊的情况下，有的系统可能有特殊的 RAM 空间结构，此时可以通过引导命令行中的选择项来改变存储空间的逻辑结构，使其正确反映内存的物理结构。此函数的作用就是分析命令行中的选择项，并据此对数据结构 e820 中的内容作出修正，其代码也在 setup.c 中。

宏定义：

```
#define PFN_UP (x)        ( ( (x)  + PAGE_SIZE-1)  >> PAGE_SHIFT)
#define PFN_DOWN (x)      ( (x)  >> PAGE_SHIFT)
#define PFN_PHYS (x)      ( (x)  << PAGE_SHIFT)
```

PFN_UP () 和 PFN_DOWN () 都是将地址 x 转换为页面号（PFN 即 Page Frame Number 的缩写），二者之间的区别为：PFN_UP () 返回大于 x 的第 1 个页面号，而 PFN_DOWN () 返回小于 x 的第 1 个页面号。宏 PFN_PHYS () 返回页面号 x 的物理地址。

宏定义

```
/*
* 128MB for vmalloc and initrd
*/
#define VMALLOC_RESERVE  (unsigned long)  (128 << 20)
#define MAXMEM  (unsigned long)  (-PAGE_OFFSET-VMALLOC_RESERVE)
#define MAXMEM_PFN PFN_DOWN (MAXMEM)
#define MAX_NONPAE_PFN  (1 << 20)
```

对这几个宏描述如下：

- VMALLOC_RESERVE：为 vmalloc () 函数访问内核空间所保留的内存区，大小为 128MB。
- MAXMEM：内核能够直接映射的最大 RAM 容量，为 1GB－128MB＝896MB（-PAGE_OFFSET 就等于 1GB）
- MAXMEM_PFN：返回由内核能直接映射的最大物理页面数。
- MAX_NONPAE_PFN：给出在 4GB 之上第 1 个页面的页面号。当页面扩充（PAE）功能启用时，才能访问 4GB 以上的内存。

获得内核映像之后的起始页面号：

```
/*
* partially used pages are not usable - thus
* we are rounding upwards:
*/
start_pfn = PFN_UP (__pa (&_end) ) ;
```

在上一节已说明，宏__pa () 返回给定虚拟地址的物理地址。其中标识符_end 表示内核映像在内核空间的结束位置。因此，存放在变量 start_pfn 中的值就是紧接着内核映像之后的页面号。

找出可用的最高页面号：

```
/*
* Find the highest page frame number we have available
*/
max_pfn = 0;
for  (i = 0; i < e820.nr_map; i++)  {
     unsigned long start, end;
     /* RAM? */
     if  (e820.map[i].type != E820_RAM)
          continue;
     start = PFN_UP (e820.map[i].addr) ;
     end = PFN_DOWN (e820.map[i].addr + e820.map[i].size) ;
     if  (start >= end)
          continue;
     if  (end > max_pfn)
          max_pfn = end;
}
```

上面这段代码循环查找类型为 E820_RAM（可用 RAM）的内存区，并把最后一个页面的页面号存放在 max_pfn 中。

确定最高和最低内存范围：

```
/*
* Determine low and high memory ranges:
*/
max_low_pfn = max_pfn;
if  (max_low_pfn > MAXMEM_PFN)  {
max_low_pfn = MAXMEM_PFN;
#ifndef CONFIG_HIGHMEM
/* Maximum memory usable is what is directly addressable */
printk (KERN_WARNING "Warning only %ldMB will be used.\n",
                                             MAXMEM>>20) ;
     if  (max_pfn > MAX_NONPAE_PFN)
          printk (KERN_WARNING "Use a PAE enabled kernel.\n") ;
     else
          printk (KERN_WARNING "Use a HIGHMEM enabled kernel.\n") ;
#else /* !CONFIG_HIGHMEM */
#ifndef CONFIG_X86_PAE
     if  (max_pfn > MAX_NONPAE_PFN)  {
          max_pfn = MAX_NONPAE_PFN;
          printk (KERN_WARNING "Warning only 4GB will be used.\n") ;
           printk (KERN_WARNING "Use a PAE enabled kernel.\n") ;
}
#endif /* !CONFIG_X86_PAE */
#endif /* !CONFIG_HIGHMEM */
}
```

有两种情况：

- 如果物理内存 RAM 大于 896MB,而小于 4GB,则选用 CONFIG_HIGHMEM 选项来进行访问；
- 如果物理内存 RAM 大于 4GB，则选用 CONFIG_X86_PAE（启用 PAE 模式）来进行访问。

上面这段代码检查了这两种情况，并显示适当的警告信息。

```
#ifdef CONFIG_HIGHMEM
highstart_pfn = highend_pfn = max_pfn;
```

```
if  (max_pfn > MAXMEM_PFN)  {
    highstart_pfn = MAXMEM_PFN;
    printk (KERN_NOTICE "%ldMB HIGHMEM available.\n",
        pages_to_mb (highend_pfn - highstart_pfn) ) ;
}
#endif
```

如果使用了 CONFIG_HIGHMEM 选项，上面这段代码仅仅打印出大于 896MB 的可用物理内存数量。

初始化引导时的分配器

```
* Initialize the boot-time allocator  (with low memory only) :
*/
bootmap_size = init_bootmem (start_pfn, max_low_pfn) ;
```

通过调用 init_bootmem（）函数，为物理内存页面管理机制的建立做初步准备，为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方，也就是说，把内核映像终点_end 上方的若干页面用作物理页面位图。在前面的代码中已经搞清楚了物理内存顶点所在的页面号为 max_low_pfn，所以物理内存的页面号一定在 0～max_low_pfn 之间。可是，在这个范围内可能有空洞（hole），另一方面，并不是所有的物理内存页面都可以动态分配。建立这个位图的目的就是要搞清楚哪一些物理内存页面可以动态分配的。后面会具体描述 bootmem 分配器。

用 bootmem 分配器，登记全部低区（0～896MB）的可用 RAM 页面

```
/*
* Register fully available low RAM pages with the
* bootmem allocator.
*/
for  (i = 0; i < e820.nr_map; i++)  {
    unsigned long curr_pfn, last_pfn, size;
/*
* Reserve usable low memory
*/
 if  (e820.map[i].type != E820_RAM)
    continue;
/*
* We are rounding up the start address of usable memory:
*/
    curr_pfn = PFN_UP (e820.map[i].addr) ;
    if  (curr_pfn >= max_low_pfn)
        continue;
/*
* ... and at the end of the usable range downwards:
*/
last_pfn = PFN_DOWN (e820.map[i].addr + e820.map[i].size) ;
    if  (last_pfn > max_low_pfn)
        last_pfn = max_low_pfn;
/*
* .. finally, did all the rounding and playing
* around just make the area go away?
*/
    if  (last_pfn <= curr_pfn)
```

```
                continue;
            size = last_pfn - curr_pfn;
            free_bootmem（PFN_PHYS（curr_pfn），PFN_PHYS（size））;
        }
```

这个循环仔细检查所有可以使用的 RAM，并调用 free_bootmem()函数把这些可用 RAM 标记为可用。这个函数调用以后，只有类型为 1（可用 RAM）的内存被标记为可用的，参看后面对这个函数的具体描述。

保留内存：

```
        /*
        * Reserve the bootmem bitmap itself as well. We do this in two
        * steps （first step was init_bootmem（））because this catches
        * the （very unlikely） case of us accidentally initializing the
        * bootmem allocator with an invalid RAM area.
        */
        reserve_bootmem（HIGH_MEMORY, （PFN_PHYS（start_pfn） +
                bootmap_size + PAGE_SIZE-1） - （HIGH_MEMORY））;
```

这个函数把内核和 bootmem 位图所占的内存标记为“保留”。 HIGH_MEMORY 为 1MB，即内核开始的地方，后面还要对这个函数进行具体描述。

分页机制的初始化

```
        paging_init（）;
```

这个函数初始化分页内存管理所需要的数据结构，参见后面的详细描述。

2．setup_memory_region() 函数

这个函数用来处理 BIOS 的内存构成图，并把这个构成图拷贝到全局变量 e820 中。如果操作失败，就创建一个伪内存构成图。这个函数的主要操作如下所述。

- 调用 sanitize_e820_map()函数，以删除内存构成图中任何重叠的部分，因为 BIOS 所报告的内存构成图可能有重叠。
- 调用 copy_e820_map()进行实际的拷贝。
- 如果操作失败，创建一个伪内存构成图，这个伪构成图有两部分：0 到 640K 及 1M 到最大物理内存。
- 打印最终的内存构成图。

3．copy_e820_map() 函数

函数原型为：

```
    static int __init sanitize_e820_map（struct e820entry * biosmap, char * pnr_map）
```

其主要操作如下概述。

(1) 如果物理内存区间小于 2，那肯定出错。因为 BIOS 至少和 RAM 属于不同的物理区间。

```
if （nr_map < 2）
        return -1;
```

(2) 从 BIOS 构成图中读出一项。

```
    do {
        unsigned long long start = biosmap->addr;
        unsigned long long size = biosmap->size;
        unsigned long long end = start + size;
```

```
        unsigned long type = biosmap->type;
```

(3) 进行检查。

```
    /* Overflow in 64 bits? Ignore the memory map. */
        if (start > end)
            return -1;
```

(4) 一些 BIOS 把 640KB～1MB 之间的区间作为 RAM 来用，这是不符合常规的。因为从 0xA0000 开始的空间用于图形卡，因此，在内存构成图中要进行修正。如果一个区的起点在 0xA0000 以下，而终点在 1MB 之上，就要将这个区间拆开成两个区间，中间跳过从 0xA0000 到 1MB 边界之间的那一部分。

```
/*
          * Some BIOSes claim RAM in the 640k - 1M region.
          * Not right. Fix it up.
          */
               if (type == E820_RAM) {
                    if (start < 0x100000ULL && end > 0xA0000ULL) {
                         if (start < 0xA0000ULL)
                              add_memory_region (start, 0xA0000ULL-start, type)
                          if (end <= 0x100000ULL)
                              continue;
                         start = 0x100000ULL;
                          size = end - start;
                    }
               }

               add_memory_region (start, size, type) ;
          } while (biosmap++,--nr_map) ;
          return 0;
```

4. add_memory_region() 函数

这个函数的功能就是在 e820 中增加一项，其主要操作如下所述。

(1) 获得已追加在 e820 中的内存区数。

```
    int x = e820.nr_map;
```

(2) 如果数目已达到最大（32），则显示一个警告信息并返回。

```
    if (x == E820MAX) {
        printk (KERN_ERR "Oops! Too many entries in
                                             the memory map!\n") ;
        return;
    }
```

(3) 在 e820 中增加一项，并给 nr_map 加 1。

```
    e820.map[x].addr = start;
    e820.map[x].size = size;
    e820.map[x].type = type;
    e820.nr_map++;
```

5. print_memory_map() 函数

这个函数把内存构成图在控制台上输出，函数本身比较简单，在此给出一个运行实例。例如函数的输出为（BIOS 所提供的物理 RAM 区间）：

```
BIOS-e820: 0000000000000000 - 00000000000a0000  (usable)
BIOS-e820: 00000000000f0000 - 0000000000100000  (reserved)
BIOS-e820: 0000000000100000 - 000000000c000000  (usable)
BIOS-e820: 00000000ffff0000 - 0000000100000000  (reserved)
```

6.2.3　物理内存的描述

为了对内存的初始化内容进行进一步的讨论，我们首先要了解 Linux 对物理内存的描述机制。

1．一致存储结构（UMA）和非一致存储结构（NUMA）

在传统的计算机结构中，整个物理内存都是均匀一致的，CPU 访问这个空间中的任何一个地址所需要的时间都相同，所以把这种内存称为"一致存储结构（Uniform Memory Architecture）"，简称 UMA。可是，在一些新的系统结构中，特别是多 CPU 结构的系统中，物理存储空间在这方面的一致性却成了问题。这是因为，在多 CPU 结构中，系统中只有一条总线（例如，PCI 总线），有多个 CPU 模块连接在系统总线上，每个 CPU 模块都有本地的物理内存，但是也可以通过系统总线访问其他 CPU 模块上的内存。另外，系统总线上还连接着一个公用的存储模块，所有的 CPU 模块都可以通过系统总线来访问它。因此，所有这些物理内存的地址可以互相连续而形成一个连续的物理地址空间。

显然，就某个特定的 CPU 而言，访问其本地的存储器速度是最快的，而穿过系统总线访问公用存储模块或其他 CPU 模块上的存储器就比较慢，而且还面临因可能的竞争而引起的不确定性。也就是说，在这样的系统中，其物理存储空间虽然地址连续，但因为所处"位置"不同而导致的存取速度不一致，所以称为"非一致存储结构（Non-Uniform Memory Architecture），简称 NUMA。

事实上，严格意义上的 UMA 结构几乎不存在。就拿配置最简单的单 CPU 来说，其物理存储空间就包括了 RAM、ROM（用于 BIOS），还有图形卡上的静态 RAM。但是，在 UMA 中，除主存 RAM 之外的存储器空间都很小，因此可以把它们放在特殊的地址上，在编程时加以特别注意就行，那么，可以认为以 RAM 为主体的主存是 UMA 结构。

由于 NUMA 的引入，就需要存储管理机制的支持，因此，Linux 内核从 2.4 版本开始就提供了对 NUMA 的支持（作为一个编译可选项）。为了对 NUMA 进行描述，引入一个新的概念——"存储节点(或叫节点)"，把访问时间相同的存储空间就叫做一个"存储节点"。一般来说，连续的物理页面应该分配在相同的存储节点上。例如，如果 CPU 模块 1 要求分配 5 个页面，但是由于本模块上的存储空间已经不够，只能分配 3 个页面，那么此时，是把另外两个页面分配在其他 CPU 模块上呢，还是把 5 个页面干脆分配在一个模块上？显然，合理的分配方式因该是将这 5 个页面都分配在公用模块上。

Linux 把物理内存划分为 3 个层次来管理 存储节点(Node)、管理区(Zone)和页面(Page)，并用 3 个相应的数据结构来描述。

2．页面（Page）数据结构

对一个物理页面的描述在／include/linux/mm.h 中：

```
/*
  * Each physical page in the system has a struct page associated with
  * it to keep track of whatever it is we are using the page for at the
  * moment. Note that we have no way to track which tasks are using
  * a page.
  *
  * Try to keep the most commonly accessed fields in single cache lines
  * here （16 bytes or greater）．  This ordering should be particularly
  * beneficial on 32-bit processors.
  *
  * The first line is data used in page cache lookup, the second line
  * is used for linear searches （eg. clock algorithm scans）．
  *
  * TODO: make this structure smaller, it could be as small as 32 bytes.
  */

typedef struct page {
        struct list_head list;          /* ->mapping has some page lists. */
         struct address_space *mapping;  /* The inode （or ...） we belong to. */
         unsigned long index;            /* Our offset within mapping. */
         struct page *next_hash;         /* Next page sharing our hash bucket in
                                            the pagecache hash table. */
         atomic_t count;                 /* Usage count, see below. */
         unsigned long flags;            /* atomic flags, some possibly
                                            updated asynchronously */
         struct list_head lru;           /* Pageout list, eg. active_list;
                                            protected by pagemap_lru_lock !! */
         wait_queue_head_t wait;         /* Page locked?  Stand in line... */
         struct page **pprev_hash;       /* Complement to *next_hash. */
         struct buffer_head * buffers;   /* Buffer maps us to a disk block. */
         void *virtual;                  /* Kernel virtual address （NULL if
                                            not kmapped, ie. highmem） */
         struct zone_struct *zone;       /* Memory zone we are in. */
} mem_map_t;
extern mem_map_t * mem_map；
```

源代码的注释中对这个数据结构给出了一定的说明，从中我们可以对此结构有一定的理解，后面还要对此结构中的每个域给出具体的解释。

内核中用来表示这个数据结构的变量常常是 page 或 map。

当页面的数据来自一个文件时，index 代表着该页面中的数据在文件中的偏移量；当页面的内容被换出到交换设备上，则 index 指明了页面的去向。结构中各个成分的次序是有讲究的，尽量使得联系紧密的若干域存放在一起，这样当这个数据结构被装入到高速缓存中时，联系紧密的域就可以存放在同一缓冲行（Cache Line）中。因为同一缓冲行（其大小为 16 字节）中的内容几乎可以同时存取，因此，代码注释中希望这个数据结构尽量地小到用 32 个字节可以描述。

系统中的每个物理页面都有一个 Page（或 mem_map_t）结构。系统在初始化阶段根据内

存的大小建立起一个 Page 结构的数组 mem_map，数组的下标就是内存中物理页面的序号。

3．管理区 Zone

为了对物理页面进行有效的管理，Linux 又把物理页面划分为 3 个区：

- 专供 DMA 使用的 ZONE_DMA 区（小于 16MB）；
- 常规的 ZONE_NORMAL 区（大于 16MB 小于 896MB）；
- 内核不能直接映射的区 ZONE_HIGME 区（大于 896MB）。

这里进一步说明为什么对 DMA 要单独设置管理区。首先，DMA 使用的页面是磁盘 I／O 所需的，如果在页面的分配过程中，所有的页面全被分配完，那么页面及盘区的交换就无法进行了，这是操作系统决不允许出现的现象。另外，在 i386 CPU 中，页式存储管理的硬件支持是在 CPU 内部实现的，而不像有些 CPU 那样由一个单独的 MMU 来提供，所以 DMA 对内存的访问不经过 MMU 提供的地址映射。这样，外部设备就要直接访问物理页面的地址。可是，有些外设（特别是插在 ISA 总线上的外设接口卡）在这方面往往有些限制，要求用于 DMA 的物理地址不能过高。另一方面，当 DMA 所需的缓冲区超过一个物理页面的大小时，就要求两个物理页面在物理上是连续的，但因为此时 DMA 控制器不能依靠 CPU 内部的 MMU 将连续的虚存页面映射到物理上也连续的页面上，因此，用于 DMA 的物理页面必须加以单独管理。

关于管理区的数据结构 zone_struct(或 zone_t)将在后面进行描述。

4．存储节点（Node）的数据结构

存储节点的数据结构为 pglist_data，定义于 Include/linux/mmzone.h 中：

```
typedef struct pglist_data {
        zone_t node_zones[MAX_NR_ZONES];
        zonelist_t node_zonelists[GFP_ZONEMASK+1];
        int nr_zones;
        struct page *node_mem_map;
        unsigned long *valid_addr_bitmap;
        struct bootmem_data *bdata;
        unsigned long node_start_paddr;
        unsigned long node_start_mapnr;
        unsigned long node_size;
        int node_id;
        struct pglist_data *node_next;
   } pg_data_t;
```

显然,若干存储节点的 pglist_data 数据结构可以通过 node_next 形成一个单链表队列。每个结构中的 node_mem_map 指向具体节点的 page 结构数组，而数组 node_zone[]就是该节点的最多 3 个页面管理区。

在 pglist_data 结构里设置了一个 node_zonelists 数组，其类型定义也在同一文件中：

```
typedef struct zonelist_struct {
     zone_t *zone[MAX_NR_ZONE+1];  //NULL delimited
     Int gfp_mast;
   } zonelist_t
```

这里的 zone[]是个指针数组，各个元素按特定的次序指向具体的页面管理区，表示分配页面时先试 zone[0]所指向的管理区,如果不能满足要求就试 zone[1]所指向的管理区,等等。

这些管理区可以属于不同的存储节点。关于管理区的分配可以有很多种策略，例如，CPU 模块 1 需要分配 5 个用于 DMA 的页面，可是它的 ZONE_DMA 只有 3 个页面，于是就从公用模块的 ZONE_DMA 中分配 5 个页面。就是说，每个 zonelist_t 规定了一种分配策略。然而，每个存储节点不应该只有一种分配策略，所以在 pglist_data 中提供的是一个 zonelist_t 数组，数组的大小 NR_GFPINDEX 为 100。

6.2.4 页面管理机制的初步建立

为了对页面管理机制作出初步准备，Linux 使用了一种叫 bootmem 分配器(Bootmem Allocator)的机制，这种机制仅仅用在系统引导时，它为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方，也就是说，内核映像终点_end 上方的地方。这个位图用来管理低区（例如小于 896MB），因为在 0 到 896MB 的范围内，有些页面可能保留，有些页面可能有空洞，因此，建立这个位图的目的就是要搞清楚哪一些物理页面是可以动态分配的。用来存放位图的数据结构为 bootmem_data（在 mm/numa.c 中）　：

```
typedef struct bootmem_data {
    unsigned long node_boot_start;
    unsigned long node_low_pfn;
    void *node_bootmem_map;
    unsigned long last_offset;
    unsigned long last_pos;
} bootmem_data_t;
```

node_boot_start 表示存放 bootmem 位图的第一个页面（即内核映像结束处的第一个页面)。

node_low_pfn 表示物理内存的顶点，最高不超过 896MB。

node_bootmem_map 指向 bootmem 位图

last_offset 用来存放在前一次分配中所分配的最后一个字节相对于 last_pos 的位移量。

last_pos 用来存放前一次分配的最后一个页面的页面号。这个域用在__alloc_bootmem_core()函数中，通过合并相邻的内存来减少内部碎片。

下面介绍与 bootmem 相关的几个函数，这些函数位于 mm/bootmeme.c 中。

1. init_bootmem()函数

```
unsigned long __init init_bootmem  (unsigned long start, unsigned long pages)
{
        max_low_pfn = pages;
        min_low_pfn = start;
        return (init_bootmem_core (&contig_page_data, start, 0, pages) ) ;
}
```

这个函数仅在初始化时用来建立 bootmem 分配器。这个函数实际上是 init_bootmem_core()函数的封装函数。init_bootmem () 函数的参数 start 表示内核映像结束处的页面号，而 pages 表示物理内存顶点所在的页面号。而函数 init_bootmem_core()就是对 contig_page_data 变量进行初始化。下面我们来看一下对该变量的定义：

```
int numnodes = 1;       /* Initialized for UMA platforms */

static bootmem_data_t contig_bootmem_data;
pg_data_t contig_page_data = { bdata: &contig_bootmem_data };
```

变量 contig_page_data 的类型就是前面介绍过的 pg_data_t 数据结构。每个 pg_data_t 数据结构代表着一片均匀的、连续的内存空间。在连续空间 UMA 结构中，只有一个节点 contig_page_data，而在 NUMA 结构或不连续空间 UMA 结构中，有多个这样的数据结构。系统中各个节点的 pg_data_t 数据结构通过 node_next 连接在一起成为一个链。有一个全局量 pgdat_list 则指向这个链。从上面的定义可以看出,contig_page_data 是链中的第一个节点。这里假定整个物理空间为均匀的、连续的，以后若发现这个假定不能成立，则将新的 pg_data_t 结构加入到链中。

pg_data_t 结构中有个指针 bdata，contig_page_data 被初始化为指向 bootmem_data_t 数据结构。下面我们来看 init_bootmem_core（）函数的具体代码：

```
/*
  * Called once to set up the allocator itself.
    */
static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
        unsigned long mapstart, unsigned long start, unsigned long end)
{
        bootmem_data_t *bdata = pgdat->bdata;
        unsigned long mapsize = ( (end - start) +7) /8;

        pgdat->node_next = pgdat_list;
        pgdat_list = pgdat;

        mapsize = (mapsize + (sizeof (long) - 1UL) ) & ~ (sizeof (long) - 1UL) ;
        bdata->node_bootmem_map = phys_to_virt (mapstart << PAGE_SHIFT) ;
        bdata->node_boot_start = (start << PAGE_SHIFT) ;
        bdata->node_low_pfn = end;

       /*
         * Initially all pages are reserved - setup_arch ()  has to
         * register free RAM areas explicitly.
        */
        memset (bdata->node_bootmem_map, 0xff, mapsize) ;

        return mapsize;
}
```

下面对这一函数给予说明。

- 变量 mapsize 存放位图的大小。(end - start)给出现有的页面数，再加个 7 是为了向上取整，除以 8 就获得了所需的字节数（因为每个字节映射 8 个页面)。
- 变量 pgdat_list 用来指向节点所形成的循环链表首部，因为只有一个节点，因此使 pgdat_list 指向自己。
- 接下来的一句使 memsize 成为下一个 4 的倍数（4 为 CPU 的字长)。例如，假设有 40 个物理页面，因此，我们可以得出 memsize 为 5 个字节。所以，上面的操作就变为（5＋（4－1)）&~(4-1)即(00001000&11111100)，最低的两位变为 0，其结果为 8。这就有效地使

memsize 变为 4 的倍数。

- phys_to_virt(mapstart << PAGE_SHIFT)把给定的物理地址转换为虚地址。
- 用节点的起始物理地址初始化 node_boot_start（这里为 0x00000000）。
- 用物理内存节点的页面号初始化 node_low_pfn。
- 初始化所有被保留的页面，即通过把页面中的所有位都置为 1 来标记保留的页面。
- 返回位图的大小。

2. free_bootmem( )函数

这个函数把给定范围的页面标记为空闲（即可用），也就是，把位图中某些位清 0，表示相应的物理内存可以投入分配。

原函数为：

```
void __init free_bootmem (unsigned long addr, unsigned long size)
{

    return (free_bootmem_core (contig_page_data.bdata, addr, size) ) ;
}
```

从上面可以看出，free_bootmem()是个封装函数，实际的工作是由 free_bootmem_core()函数完成的：

```
static void __init free_bootmem_core (bootmem_data_t *bdata, unsigned long addr, unsigned
long size)
{
        unsigned long i;
        unsigned long start;
        /*
         * round down end of usable mem, partially free pages are
        * considered reserved.
         */
        unsigned long sidx;
        unsigned long eidx = (addr + size - bdata->node_boot_start) /PAGE_SIZE;
        unsigned long end = (addr + size) /PAGE_SIZE;

        if (!size) BUG () ;
          if (end > bdata->node_low_pfn)
                BUG () ;

          /*
         * Round up the beginning of the address.
         */
        start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
        sidx = start - (bdata->node_boot_start/PAGE_SIZE) ;

        for (i = sidx; i < eidx; i++) {
                if (!test_and_clear_bit (i, bdata->node_bootmem_map) )
                        BUG () ;
        }
}
```

对此函数的解释如下。

- 变量 eidx 被初始化为页面总数。
- 变量 end 被初始化为最后一个页面的页面号。
- 进行两个可能的条件检查。
- start 初始化为第一个页面的页面号（向上取整），而 sidx(start index)初始化为相对于 node_boot_start 的页面号。
- 清位图中从 sidx 到 eidx 的所有位，即把这些页面标记为可用。

3．reserve_bootmem()函数

这个函数用来保留页面。为了保留一个页面，只需要在 bootmem 位图中把相应的位置为 1 即可。

原函数为：

```
void __init reserve_bootmem (unsigned long addr, unsigned long size)
 {
       reserve_bootmem_core (contig_page_data.bdata, addr, size) ;
 }
```

reserve_bootmem（）为封装函数，实际调用的是 reserve_bootmem_core（）函数：

```
static void __init reserve_bootmem_core (bootmem_data_t *bdata, unsigned long addr, unsigned long size)
{
        unsigned long i;
        /*
         * round up, partially reserved pages are considered
         * fully reserved.
        */
        unsigned long sidx = (addr - bdata->node_boot_start) /PAGE_SIZE;
        unsigned long eidx = (addr + size - bdata->node_boot_start +
                                                PAGE_SIZE-1) /PAGE_SIZE;
        unsigned long end = (addr + size + PAGE_SIZE-1) /PAGE_SIZE;

        if (!size) BUG () ;

        if (sidx < 0)
                BUG () ;
        if (eidx < 0)
                BUG () ;
        if (sidx >= eidx)
                BUG () ;
        if ( (addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
                BUG () ;
        if (end > bdata->node_low_pfn)
                BUG () ;
        for (i = sidx; i < eidx; i++)
                if (test_and_set_bit (i, bdata->node_bootmem_map) )
                        printk ("hm, page %08lx reserved twice.\n", i*PAGE_SIZE) ;
}
```

对此函数的解释如下。

- sidx (start index)初始化为相对于 node_boot_start 的页面号。

- 变量 eidx 初始化为页面总数（向上取整）。
- 变量 end 初始化为最后一个页面的页面号（向上取整）。
- 进行各种可能的条件检查。
- 把位图中从 sidx 到 eidx 的所有位置 1。

4．__alloc_bootmem()函数

这个函数以循环轮转的方式从不同节点分配页面。因为在 i386 上只有一个节点，因此只循环一次。

函数原型为：

```
void * __alloc_bootmem (unsigned long size,
                        unsigned long align,
                        unsigned long goal) ;
void * __alloc_bootmem_core (bootmem_data_t *bdata,
                             unsigned long size,
                             unsigned long align,
                             unsigned long goal) ;
```

其中__alloc_bootmem()为封装函数，实际调用的函数为__alloc_bootmem_core （），因为__alloc_bootmem_core （）函数比较长，下面分片断来进行仔细分析。

```
unsigned long i, start = 0;
void *ret;
unsigned long offset, remaining_size;
unsigned long areasize, preferred, incr;
unsigned long eidx = bdata->node_low_pfn -
                (bdata->node_boot_start >> PAGE_SHIFT) ;
```

把 eidx 初始化为本节点中现有页面的总数。

```
if (!size) BUG () ;
if (align & (align-1) )
    BUG () ;
```

进行条件检查。

```
/*
* We try to allocate bootmem pages above 'goal'
* first, then we try to allocate lower pages.
*/
if (goal && (goal >= bdata->node_boot_start) &&
          ( (goal >> PAGE_SHIFT) < bdata->node_low_pfn) ) {
    preferred = goal - bdata->node_boot_start;
} else
preferred = 0;
preferred = ( (preferred + align - 1) & ~ (align - 1) ) >> PAGE_SHIFT;
```

开始分配后首选页的计算分为两步：

(1) 如果 goal 为非 0 且有效，则给 preferred 赋初值，否则，其初值为 0。

(2) 根据参数 align 来对齐 preferred 的物理地址。

```
areasize = (size+PAGE_SIZE-1) /PAGE_SIZE;
```

获得所需页面的总数（向上取整）

```
incr = align >> PAGE_SHIFT ? : 1;
```

根据对齐的大小来选择增加值。除非大于 4KB（很少见），否则增加值为 1。

```
restart_scan:
    for (i = preferred; i < eidx; i += incr) {
    unsigned long j;
       if (test_bit (i, bdata->node_bootmem_map) )
         continue;
```

这个循环用来从首选页面号开始，找到空闲的页面号。test_bit()宏用来测试给定的位，如果给定位为 1，则返回 1。

```
for (j = i + 1; j < i + areasize; ++j) {
    if (j >= eidx)
        goto fail_block;
    if (test_bit (j, bdata->node_bootmem_map) )
       goto fail_block;
}
```

这个循环用来查看在首次满足内存需求以后，是否还有足够的空闲页面。如果没有空闲页，就跳到 *fail_block*。

```
start = i;
    goto found;
```

如果一直到了这里，则说明从 *i* 开始找到了足够的页面，跳过 fail_block 并继续。

```
fail_block:;
   }
 if (preferred) {
    preferred = 0;
    goto restart_scan;
       }
 return NULL;
```

如果到了这里，从首选页面中没有找到满足需要的连续页面，就忽略 preferred 的值，并从 0 开始扫描。如果 preferred 为 1，但没有找到满足需要的足够页面，则返回 NULL。

```
found:
```

已经找到足够的内存，继续处理请求。

```
if (start >= eidx)
    BUG () ;
```

进行条件检查。

```
/*
* Is the next page of the previous allocation-end the start
* of this allocation's buffer? If yes then we can 'merge'
* the previous partial page with this allocation.
*/
if (align <= PAGE_SIZE && bdata->last_offset
                     && bdata->last_pos+1 == start) {
    offset = (bdata->last_offset+align-1) & ~ (align-1) ;
    if (offset > PAGE_SIZE)
        BUG () ;
    remaining_size = PAGE_SIZE-offset;
```

if 语句检查下列条件：

(1) 所请求对齐的值小于页的大小（4KB）。

(2) 变量 last_offset 为非 0。如果为 0，则说明前一次分配达到了一个非常好的页面边界，没有内部碎片。

（3）检查这次请求的内存是否与前一次请求的内存是相临的，如果是，则把两次分配合在一起进行。

如果以上 3 个条件都满足，则用前一次分配中最后一页剩余的空间初始化 remaining_size。

```
if  (size < remaining_size)  {
    areasize = 0;
    // last_pos unchanged
    bdata->last_offset = offset+size;
    ret = phys_to_virt (bdata->last_pos*PAGE_SIZE
                + offset + bdata->node_boot_start) ;
```

如果请求内存的大小小于前一次分配中最后一页中的可用空间，则没必要分配任何新的页。变量 last_offset 增加到新的偏移量，而 last_pos 保持不变，因为没有增加新的页。把这次新分配的起始地址存放在变量 ret 中。宏 phys_to_virt()返回给定物理地址的虚地址。

```
} else {
    remaining_size = size - remaining_size;
    areasize =  (remaining_size+PAGE_SIZE-1) /PAGE_SIZE;
    ret = phys_to_virt (bdata->last_pos*PAGE_SIZE
                 + offset + bdata->node_boot_start) ;
    bdata->last_pos = start+areasize-1;
    bdata->last_offset = remaining_size;
```

所请求的大小大于剩余的大小。首先求出所需的页面数，然后更新变量 last_pos 和 last_offset。

例如，在前一次分配中，如果分配了 9KB，则占用 3 个页面，内部碎片为 12KB-9KB=3KB。因此，page_offset 为 1KB，且剩余大小为 3KB。如果新的请求为 1KB，则第 3 个页面本身就能满足要求，但是，如果请求的大小为 10KB，则需要新分配((10KB- 3KB) + PAGE_SIZE-1)/PAGE_SIZE，即 2 个页面，因此，page_offset 为 3KB。

```
    }
    bdata->last_offset &= ~PAGE_MASK;
} else {
    bdata->last_pos = start + areasize - 1;
    bdata->last_offset = size & ~PAGE_MASK;
    ret = phys_to_virt (start * PAGE_SIZE +
                              bdata->node_boot_start) ;
   }
```

如果因为某些条件未满足而导致不能进行合并，则执行这段代码，我们刚刚把 last_pos 和 last_offset 直接设置为新的值，而未考虑它们原先的值。last_pos 的值还要加上所请求的页面数，而新 page_offset 值的计算就是屏蔽掉除了获得页偏移量位的所有位，即“ size & PAGE_MASK ”， PAGE_MASK 为 0x00000FFF，用 PAGE_MASK 的求反正好得到页的偏移量。

```
/*
* Reserve the area now:
*/

for  (i = start; i < start+areasize; i++)
     if  (test_and_set_bit (i, bdata->node_bootmem_map) )
          BUG () ;
memset (ret, 0, size) ;
```

```
return ret;
```

现在，我们有了内存，就需要保留它。宏 test_and_set_bit()用来测试并置位，如果某位原先的值为 0，则它返回 0；如果为 1，则返回 1。还有一个条件判断语句，进行条件判断(这种条件出现的可能性非常小，除非 RAM 坏)。然后，把这块内存初始化为 0，并返回给调用它的函数。

5．free_all_bootmem()函数

这个函数用来在引导时释放页面，并清除 bootmem 分配器。

函数原型为：

```
void free_all_bootmem (void) ;
void free_all_bootmem_core (pg_data_t *pgdat) ;
```

同前面的函数调用形式类似，free_all_bootmem（）为封装函数，实际调用 free_all_bootmem_core（）函数。下面，我们对 free_all_bootmem_core（）函数分片断来介绍。

```
struct page *page = pgdat->node_mem_map;
bootmem_data_t *bdata = pgdat->bdata;
unsigned long i, count, total = 0;
unsigned long idx;

if (!bdata->node_bootmem_map) BUG () ;
count = 0;
idx = bdata->node_low_pfn - (bdata->node_boot_start
                                  >> PAGE_SHIFT) ;
```

把 idx 初始化为从内核映像结束处到内存顶点处的页面数。

```
for (i = 0; i < idx; i++, page++) {
    if (!test_bit (i, bdata->node_bootmem_map) ) {
        count++;
        ClearPageReserved (page) ;
        set_page_count (page, 1) ;
        __free_page (page) ;
    }
}
```

搜索 bootmem 位图,找到空闲页,并把 mem_map 中对应的项标记为空闲。set_page_count()函数把 page 结构的 count 域置 1，而__free_page()真正的释放页面，并修改伙伴（buddy）系统的位图。

```
total += count;

/*
* Now free the allocator bitmap itself, it's not
* needed anymore:
*/
page = virt_to_page (bdata->node_bootmem_map) ;
count = 0;
for (i = 0; i < ( (bdata->node_low_pfn- (bdata->node_boot_start
             >> PAGE_SHIFT) ) /8 + PAGE_SIZE-1) /PAGE_SIZE;
                                   i++,page++) {
    count++;
```

```
        ClearPageReserved (page) ;
        set_page_count (page, 1) ;
        __free_page (page) ;
    }
```

获得 bootmem 位图的地址，并释放它所在的页面。

```
    total += count;
    bdata->node_bootmem_map = NULL;
    return total;
```

把该存储节点的 bootmem_map 域置为 NULL，并返回空闲页面的总数。

6.2.5 页表的建立

前面已经建立了为内存页面管理所需的数据结构，现在是进一步完善页面映射机制，并且建立起内存页面映射管理机制的时候了，与此相关的主要函数有：

paging_init() 函数

pagetable_init() 函数

1. paging_init() 函数

这个函数仅被调用一次，即由 setup_arch()调用以建立页表，对此函数的具体描述如下：

```
    pagetable_init () ;
```

这个函数实际上才真正地建立页表，后面会给出详细描述。

```
__asm__ ( "movl %%ecx,%%cr3\n" ::"c" (__pa (swapper_pg_dir) ) ) ;
```

因为 pagetable_init()已经建立起页表，因此把 swapper_pg_dir（页目录）的地址装入 CR3 寄存器。

```
    #if CONFIG_X86_PAE
    /*
    * We will bail out later - printk doesnt work right now so
    * the user would just see a hanging kernel.
    */
    if  (cpu_has_pae)
        set_in_cr4 (X86_CR4_PAE) ;
    #endif

    __flush_tlb_all () ;
```

上面这一句是个宏，它使得转换旁路缓冲区（TLB）无效。TLB 总是要维持几个最新的虚地址到物理地址的转换。每当页目录改变时，TLB 就需要被刷新。

```
    #ifdef CONFIG_HIGHMEM
    kmap_init () ;
    #endif
```

如果使用了 CONFIG_HIGHMEM 选项，就要对大于 896MB 的内存进行初始化，我们不准备对这部分内容进行详细讨论。

```
    {
        unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
        unsigned int max_dma, high, low;
```

```
            max_dma = virt_to_phys ( (char *) MAX_DMA_ADDRESS)
                                    >> PAGE_SHIFT;
```

低于 16MB 的内存只能用于 DMA，因此，上面这条语句用于存放 16MB 的页面。

```
            low = max_low_pfn;
            high = highend_pfn;

        if  (low < max_dma)
            zones_size[ZONE_DMA] = low;
        else {
            zones_size[ZONE_DMA] = max_dma;
            zones_size[ZONE_NORMAL] = low - max_dma;
        #ifdef CONFIG_HIGHMEM
            zones_size[ZONE_HIGHMEM] = high - low;
        #endif
          }
```

计算 3 个管理区的大小，并存放在 zones_size 数组中。3 个管理区如下所述。

- ZONE_DMA：从 0～16MB 分配给这个区。
- ZONE_NORMAL：从 16MB～896MB 分配给这个区。
- ZONE_DMA ：896MB 以上分配给这个区。

```
            free_area_init (zones_size) ;
            }

            return;
```

这个函数用来初始化内存管理区并创建内存映射表，详细介绍参见后面内容。

2．pagetable_init()函数

这个函数真正地在页目录 swapper_pg_dir 中建立页表，描述如下：

```
        unsigned long vaddr, end;
        pgd_t *pgd, *pgd_base;
        int i, j, k;
        pmd_t *pmd;
        pte_t *pte, *pte_base;

        /*
        * This can be zero as well - no problem, in that case we exit
        * the loops anyway due to the PTRS_PER_* conditions.
        */
        end =  (unsigned long) __va (max_low_pfn*PAGE_SIZE) ;
```

计算 max_low_pfn 的虚拟地址，并把它存放在 end 中。

```
        pgd_base = swapper_pg_dir;
```

让 pgd_base（页目录基地址）指向 swapper_pg_dir。

```
        #if CONFIG_X86_PAE
        for  (i = 0; i < PTRS_PER_PGD; i++)
            set_pgd (pgd_base + i, __pgd (1 + __pa (empty_zero_page) ) ) ;
        #endif
```

如果 PAE 被激活，PTRS_PER_PGD 就为 4，且变量 swapper_pg_dir 用作页目录指针表，宏 set_pgd() 定义于 include/asm-i386/pgtable-3level.h 中。

```
        i = __pgd_offset (PAGE_OFFSET) ;
        pgd = pgd_base + i;
```

宏__pgd_offset()在给定地址的页目录中检索相应的下标。因此__pgd_offset (PAGE_OFFSET) 返回 0x300 (或 十进制 768), 即内核地址空间开始处的下标。因此, pgd 现在指向页目录表的第 768 项。

```
        for (; i < PTRS_PER_PGD; pgd++, i++) {
            vaddr = i*PGDIR_SIZE;
            if (end && (vaddr >= end) )
           break;
```

如果使用了 CONFIG_X86_PAE 选项, PTRS_PER_PGD 就为 4，否则，一般情况下它都为 1024，即页目录的项数。PGDIR_SIZE 给出一个单独的页目录项所能映射的 RAM 总量，在两级页目录中它为 4MB，当使用 CONFIG_X86_PAE 选项时，它为 1GB。计算虚地址 vaddr，并检查它是否到了虚拟空间的顶部。

```
        #if CONFIG_X86_PAE
            pmd = (pmd_t *) alloc_bootmem_low_pages (PAGE_SIZE) ;
            set_pgd (pgd, __pgd (__pa (pmd) + 0x1) ) ;
        #else
             pmd = (pmd_t *) pgd;
        #endif
```

如果使用了 CONFIG_X86_PAE 选项，则分配一页 (4KB) 的内存给 bootmem 分配器用，以保存中间页目录，并在总目录中设置它的地址。否则，没有中间页目录，就把中间页目录直接映射到总目录。

```
        if (pmd != pmd_offset (pgd, 0) )
                BUG () ;
            for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
                vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
                if (end && (vaddr >= end) )
                    break;
                 if (cpu_has_pse) {
                    unsigned long __pe;
                    set_in_cr4 (X86_CR4_PSE) ;
                    boot_cpu_data.wp_works_ok = 1;
                    __pe = _KERNPG_TABLE + _PAGE_PSE + __pa (vaddr) ;
                    /* Make it "global" too if supported */
                    if (cpu_has_pge) {
                        set_in_cr4 (X86_CR4_PGE) ;
                        __pe += _PAGE_GLOBAL;
                    }
                    set_pmd (pmd, __pmd (__pe) ) ;
                     continue;
                }
```

现在，开始填充页目录 (如果有 PAE，就是填充中间页目录)。计算表项所映射的虚地址，如果没有激活 PAE， PMD_SIZE 大小就为 0，因此，vaddr = i * 4MB。例如，表项 0x300 所映射的虚地址为 0x300 * 4MB = 3GB。接下来，我们检查 PSE (Page Size Extension)是否可用，如果是，就要避免使用页表而直接使用 4MB 的页。宏 cpu_has_pse () 用来检查处理器是否具有扩展页，如果有，则宏 set_in_cr4()就启用它。

从 Pentium II 处理器开始，就可以有附加属性 PGE（Page Global Enable）。当一个页被标记为全局的，且设置了 PGE，那么，在任务切换发生或 CR3 被装入时，就不能使该页所在的页表（或页目录项）无效。这将提高系统性能，也是让内核处于 3GB 以上的原因之一。选择了所有属性后，设置中间页目录项。

```
                pte_base = pte = (pte_t *)
                        alloc_bootmem_low_pages (PAGE_SIZE) ;
```

如果 PSE 不可用，就执行这一句，它为一个页表（4KB）分配空间。

```
                for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
                    vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
                    if (end && (vaddr >= end))
                        break;
```

在一个页表中有 1024 个表项（如果启用 PAE，就是 512 个），每个表项映射 4KB（1 页）。

```
                    *pte = mk_pte_phys (__pa (vaddr) , PAGE_KERNEL) ;
                }
```

宏 mk_pte_phys()创建一个页表项，这个页表项的物理地址为__pa(vaddr)。属性 PAGE_KERNEL 表示只有在内核态才能访问这一页表项。

```
                set_pmd (pmd, __pmd (_KERNPG_TABLE + __pa (pte_base) ) ) ;
                if (pte_base != pte_offset (pmd, 0) )
                    BUG () ;
            }
        }
```

通过调用 set_pmd()把该页表追加到中间页目录中。这个过程一直继续，直到把所有的物理内存都映射到从 PAGE_OFFSET 开始的虚拟地址空间。

```
        /*
            * Fixed mappings, only the page table structure has to be
            * created - mappings will be set by set_fixmap () :
            */
            vaddr = __fix_to_virt (__end_of_fixed_addresses - 1) & PMD_MASK;
            fixrange_init (vaddr, 0, pgd_base) ;
```

在内存的最高端(4GB～128MB)，有些虚地址直接用在内核资源的某些部分中，这些地址的映射定义在/include/asm/fixmap.h 中，枚举类型__end_of_fixed_addresses 用作索引，宏__fix_to_virt()返回给定索引的虚地址。函数 fixrange_init()为这些虚地址创建合适的页表项。注意，这里仅仅创建了页表项，而没有进行映射。这些地址的映射是由 set_fixmap()函数完成的。

```
            #if CONFIG_HIGHMEM
            /*
            * Permanent kmaps:
            */
            vaddr = PKMAP_BASE;
            fixrange_init (vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base) ;
            pgd = swapper_pg_dir + __pgd_offset (vaddr) ;
            pmd = pmd_offset (pgd, vaddr) ;
            pte = pte_offset (pmd, vaddr) ;
            pkmap_page_table = pte;
            #endif
```

如果使用了 CONFIG_HIGHMEM 选项，我们就可以访问 896MB 以上的物理内存，这些内存

的地址被暂时映射到为此目的而保留的虚地址上。PKMAP_BASE 的值为 0xFE000000（即 4064MB），LAST_PKMAP 的值为 1024。因此，从 4064MB 开始，由 fixrange_init()在页表中创建的表项能覆盖 4MB 的空间。接下来，把覆盖 4MB 内存的页表项赋给 pkmap_page_table。

```
#if CONFIG_X86_PAE
/*
* Add low memory identity-mappings - SMP needs it when
* starting up on an AP from real-mode. In the non-PAE
* case we already have these mappings through head.S.
* All user-space mappings are explicitly cleared after
* SMP startup.
*/
pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
#endif
```

6.2.6 内存管理区

前面已经提到，物理内存被划分为 3 个区来管理，它们是 ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM。每个区都用 struct zone_struct 结构来表示，定义于 include/linux/mmzone.h：

```
typedef struct zone_struct {
/*
* Commonly accessed fields:
*/
    spinlock_t lock;
    unsigned long    free_pages;
    unsigned long    pages_min, pages_low, pages_high;
    int              need_balance;

/*
* free areas of different sizes
*/
    free_area_t free_area[MAX_ORDER];

/*
* Discontig memory support fields.
*/
    struct pglist_data *zone_pgdat;
    struct page *zone_mem_map;
    unsigned long           zone_start_paddr;
    unsigned long           zone_start_mapnr;

/*
* rarely used fields:
*/
    char                    *name;
    unsigned long           size;
} zone_t;

#define ZONE_DMA                0
```

```
#define ZONE_NORMAL            1
#define ZONE_HIGHMEM           2
#define MAX_NR_ZONES           3
```

对 struct zone_struct 结构中每个域的描述如下。

- lock ：用来保证对该结构中其他域的串行访问。
- free_pages ：在这个区中现有空闲页的个数。
- pages_min、pages_low 及 pages_high 是对这个区最少、次少及最多页面个数的描述。
- need_balance：与 kswapd 合在一起使用。
- free_area：在伙伴分配系统中的位图数组和页面链表。
- zone_pgdat：本管理区所在的存储节点。
- zone_mem_map：该管理区的内存映射表。
- zone_start_paddr：该管理区的起始物理地址。
- zone_start_mapnr：在 mem_map 中的索引（或下标）。
- name：该管理区的名字。
- size：该管理区物理内存总的大小。

其中，free_area_t 定义为：

```
#difine   MAX_ORDER  10
    type struct free_area_struct {
           struct list_head   free_list
            unsigned  int    *map
     } free_area_t
```

因此，zone_struct 结构中的 free_area[MAX_ORDER]是一组“空闲区间”链表。为什么要定义一组而不是一个空闲队列呢？这是因为常常需要成块地在物理空间分配连续的多个页面，所以要按块的大小分别加以管理。因此，在管理区数据结构中既要有一个队列来保持一些离散（连续长度为 1）的物理页面，还要有一个队列来保持一些连续长度为 2 的页面块以及连续长度为 4、8、16、……、直至 2^{MAX_ORDER}（即 4M 字节）的队列。

如前所述，内存中每个物理页面都有一个 struct page 结构，位于 include/linux/mm.h，该结构包含了对物理页面进行管理的所有信息，下面给出具体描述：

```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    wait_queue_head_t wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
    void *virtual;
    struct zone_struct *zone;
} mem_map_t;
```

对每个域的描述如下。

- list ：指向链表中的下一页。

- mapping ：用来指定我们正在映射的索引节点（inode)。
- index：在映射表中的偏移。
- next_hash：指向页高速缓存哈希表中下一个共享的页。
- count：引用这个页的个数。
- flags：页面各种不同的属性。
- lru：用在 active_list 中。
- wait：等待这一页的页队列。
- pprev_hash：与 next_hash 相对应。
- buffers：把缓冲区映射到一个磁盘块。
- zone：页所在的内存管理区。

与内存管理区相关的 3 个主要函数为：

- free_area_init() 函数；
- build_zonelists() 函数；
- mem_init() 函数。

1. free_area_init() 函数

这个函数用来初始化内存管理区并创建内存映射表，定义于 mm/page_alloc.c 中。

函数原型为：

```
void free_area_init (unsigned long *zones_size) ;
void free_area_init_core (int nid, pg_data_t *pgdat,
                          struct page **gmap,
                          unsigned long *zones_size,
                          unsigned long zone_start_paddr,
                          unsigned long *zholes_size,
                          struct page *lmem_map) ;
```

free_area_init ()为封装函数，而 free_area_init_core ()为真正实现的函数，对该函数详细描述如下：

```
        struct page *p;
        unsigned long i, j;
    unsigned long map_size;
        unsigned long totalpages, offset, realtotalpages;
    const unsigned long zone_required_alignment = 1UL << (MAX_ORDER-1) ;

        if (zone_start_paddr & ~PAGE_MASK)
       BUG () ;
```

检查该管理区的起始地址是否是一个页的边界。

```
        totalpages = 0;
        for (i = 0; i < MAX_NR_ZONES; i++) {
            unsigned long size = zones_size[i];
            totalpages += size;
        }
```

计算本存储节点中页面的个数。

```
    realtotalpages = totalpages;
        if (zholes_size)
```

```
        for  (i = 0; i < MAX_NR_ZONES; i++)
            realtotalpages -= zholes_size[i];
    printk ("On node %d totalpages: %lu\n", nid, realtotalpages) ;
```

打印除空洞以外的实际页面数。

```
    INIT_LIST_HEAD (&active_list) ;
    INIT_LIST_HEAD (&inactive_list) ;
```

初始化循环链表。

```
    /*
    * Some architectures  (with lots of mem and discontinous memory
    * maps)  have to search for a good mem_map area:
    * For discontigmem, the conceptual mem map array starts from
    * PAGE_OFFSET, we need to align the actual array onto a mem map
    * boundary, so that MAP_NR works.
    */
    map_size =  (totalpages + 1) *sizeof (struct page) ;
    if  (lmem_map ==  (struct page *) 0)  {
        lmem_map =  (struct page *)
                alloc_bootmem_node (pgdat, map_size) ;
        lmem_map =  (struct page *)  (PAGE_OFFSET +
        MAP_ALIGN ( (unsigned long) lmem_map - PAGE_OFFSET) ) ;
    }
```

给局部内存（即本节点中的内存）映射分配空间，并在 sizeof(mem_map_t)边界上对齐它。

```
*gmap = pgdat->node_mem_map = lmem_map;
    pgdat->node_size = totalpages;
pgdat->node_start_paddr = zone_start_paddr;
    pgdat->node_start_mapnr =  (lmem_map - mem_map) ;
pgdat->nr_zones = 0;
```

初始化本节点中的域。

```
        /*
    * Initially all pages are reserved - free ones are freed
    * up by free_all_bootmem ()  once the early boot process is
    * done.
    */
    for  (p = lmem_map; p < lmem_map + totalpages; p++)  {
        set_page_count (p, 0) ;
        SetPageReserved (p) ;
        init_waitqueue_head (&p->wait) ;
        memlist_init (&p->list) ;
    }
```

仔细检查所有的页，并进行如下操作。

（1）把页的使用计数（count 域）置为 0。

（2）把页标记为保留。

（3）初始化该页的等待队列。

（4）初始化链表指针。

```
    offset = lmem_map - mem_map;
```

变量 mem_map 是类型为 struct pages 的全局稀疏矩阵。mem_map 下标的起始值取决于第一个节点的第一个管理区。如果第一个管理区的起始地址为 0，则下标就从 0 开始，并且与

物理页面号相对应，也就是说，页面号就是 mem_map 的下标。每一个管理区都有自己的映射表，存放在 zone_mem_map 中，每个管理区又被映射到它所在的节点 node_mem_map 中，而每个节点又被映射到管理全局内存的 mem_map 中。

在上面的这行代码中，offset 表示该节点放的内存映射表在全局 mem_map 中的入口点(下标)。在这里，offset 为 0，因为在 i386 上，只有一个节点。

```
for (j = 0; j < MAX_NR_ZONES; j++) {
```

这个循环对 zone 的域进行初始化。

```
zone_t *zone = pgdat->node_zones + j;
unsigned long mask;
unsigned long size, realsize;
realsize = size = zones_size[j];
```

管理区的实际数据是存放在节点中的，因此，让指针指向正确的管理区，并获得该管理区的大小。

```
if (zholes_size)
    realsize -= zholes_size[j];

printk("zone(%lu): %lu pages.\n", j, size);
```

计算各个区的实际大小，并进行打印。例如，在具有 256MB 的内存上，上面的输出为：

```
zone(0): 4096 pages.
zone(1): 61440 pages.
zone(2): 0 pages.
```

这里，管理区 2 为 0，因为只有 256MB 的 RAM.。

```
zone->size = size;
zone->name = zone_names[j];
zone->lock = SPIN_LOCK_UNLOCKED;
zone->zone_pgdat = pgdat;
zone->free_pages = 0;
zone->need_balance = 0;
```

初始化管理区中的各个域。

```
if (!size)
    continue;
```

如果一个管理区的大小为 0，就没必要进一步的初始化。

```
pgdat->nr_zones = j+1;
mask = (realsize / zone_balance_ratio[j]);
if (mask < zone_balance_min[j])
    mask = zone_balance_min[j];
else if (mask > zone_balance_max[j])
    mask = zone_balance_max[j];
```

计算合适的平衡比率。

```
zone->pages_min = mask;
zone->pages_low = mask*2;
zone->pages_high = mask*3;
zone->zone_mem_map = mem_map + offset;
zone->zone_start_mapnr = offset;
zone->zone_start_paddr = zone_start_paddr;
```

设置该管理区中页面数量的几个界限，并把在全局变量 mem_map 中的入口点作为

zone_mem_map 的初值。用全局变量 mem_map 的下标初始化变量 zone_start_mapnr。

```
        if ((zone_start_paddr >> PAGE_SHIFT) &
                        (zone_required_alignment-1))
            printk("BUG: wrong zone alignment, it will crash\n");

        for (i = 0; i < size; i++) {
            struct page *page = mem_map + offset + i;
            page->zone = zone;
            if (j != ZONE_HIGHMEM)
                page->virtual = __va(zone_start_paddr);
            zone_start_paddr += PAGE_SIZE;
        }
```

对该管理区中的每一页进行处理。首先，把 struct page 结构中的 zone 域初始化为指向该管理区（zone）,如果这个管理区不是 ZONE_HIGHMEM，则设置这一页的虚地址（即物理地址 + PAGE_OFFSET)。也就是说，建立起每一页物理地址到虚地址的映射。

```
        offset += size;
```

把 offset 增加 size，使它指向 mem_map 中下一个管理区的起始位置。

```
        for (i = 0; ; i++) {
            unsigned long bitmap_size;
            memlist_init(&zone->free_area[i].free_list);
            if (i == MAX_ORDER-1) {
                zone->free_area[i].map = NULL;
                break;
            }
```

初始化 free_area[]链表，把 free_area[]中最后一个序号的位图置为 NULL。

```
         /*
         * Page buddy system uses "index >> (i+1)",
         * where "index" is at most "size-1".
         *
         * The extra "+3" is to round down to byte
         * size (8 bits per byte assumption). Thus
         * we get "(size-1) >> (i+4)" as the last byte
         * we can access.
         *
         * The "+1" is because we want to round the
         * byte allocation up rather than down. So
         * we should have had a "+7" before we shifted
         * down by three. Also, we have to add one as
         * we actually _use_ the last bit (it's [0,n]
         * inclusive, not [0,n[).
         *
         * So we actually had +7+1 before we shift
         * down by 3. But (n+8) >> 3 == (n >> 3) + 1
         * (modulo overflows, which we do not have).
         *
         * Finally, we LONG_ALIGN because all bitmap
         * operations are on longs.
         */
                bitmap_size = (size-1) >> (i+4);
                bitmap_size = LONG_ALIGN(bitmap_size+1);
```

```
                zone->free_area[i].map = (unsigned long *)
                        alloc_bootmem_node (pgdat, bitmap_size) ;
        }
```

计算位图的大小，然后调用 alloc_bootmem_node 给位图分配空间。

```
        }

        build_zonelists (pgdat) ;
```

在节点中为不同的管理区创建链表。

2. build_zonelists () 函数

函数原型：

```
static inline void build_zonelists (pg_data_t *pgdat)
```

代码如下：

```
int i, j, k;

for (i = 0; i <= GFP_ZONEMASK; i++) {
    zonelist_t *zonelist;
    zone_t *zone;
    zonelist = pgdat->node_zonelists + i;
    memset (zonelist, 0, sizeof (*zonelist) ) ;
```

获得节点中指向管理区链表的域，并把它初始化为空。

```
    j = 0;
    k = ZONE_NORMAL;
    if (i & __GFP_HIGHMEM)
        k = ZONE_HIGHMEM;
    if (i & __GFP_DMA)
        k = ZONE_DMA;
```

把当前管理区掩码与 3 个可用管理区掩码相“与”，获得一个管理区标识，把它用在下面的 switch 语句中。

```
    switch (k) {
        default:
            BUG () ;
    /*
    * fallthrough:
    */
        case ZONE_HIGHMEM:
            zone = pgdat->node_zones + ZONE_HIGHMEM;
            if (zone->size) {
            #ifndef CONFIG_HIGHMEM
                BUG () ;
            #endif
                zonelist->zones[j++] = zone;
            }
        case ZONE_NORMAL:
            zone = pgdat->node_zones + ZONE_NORMAL;
            if (zone->size)
                zonelist->zones[j++] = zone;
        case ZONE_DMA:
            zone = pgdat->node_zones + ZONE_DMA;
```

```
                if (zone->size)
                    zonelist->zones[j++] = zone;
            }
```

给定的管理区掩码指定了优先顺序，我们可以用它找到在 switch 语句中的入口点。如果掩码为__GFP_DMA，管理区链表 zonelist 将仅仅包含 DMA 管理区，如果为__GFP_HIGHMEM，则管理区链表中就会依次有 ZONE_HIGHMEM、 ZONE_NORMAL 和 ZONE_DMA。

```
            zonelist->zones[j++] = NULL;
        }
```

用 NULL 结束链表。

3．mem_init() 函数

这个函数由 start_kernel()调用，以对管理区的分配算法进行进一步的初始化，定义于 arch/i386/mm/init.c 中，具体解释如下：

```
        int codesize, reservedpages, datasize, initsize;
        int tmp;
        int bad_ppro;

        if (!mem_map)
             BUG () ;

        #ifdef CONFIG_HIGHMEM
        highmem_start_page = mem_map + highstart_pfn;
        max_mapnr = num_physpages = highend_pfn;
```

如果 HIGHMEM 被激活，就要获得 HIGHMEM 的起始地址和总的页面数。

```
        #else
        max_mapnr = num_physpages = max_low_pfn;
        #endif
```

否则，页面数就是常规内存的页面数。

```
        high_memory = (void *) __va (max_low_pfn * PAGE_SIZE) ;
```

获得低区内存中最后一个页面的虚地址。

```
        /* clear the zero-page */
        memset (empty_zero_page, 0, PAGE_SIZE) ;

        /* this will put all low memory onto the freelists */
        totalram_pages += free_all_bootmem () ;
        reservedpages = 0;
```

free_all_bootmem()函数本质上释放所有的低区内存，从此以后，bootmem 不再使用。

```
        /*
        * Only count reserved RAM pages
        */
             for (tmp = 0; tmp < max_low_pfn; tmp++)
             if (page_is_ram (tmp) && PageReserved (mem_map+tmp) )
                  reservedpages++;
```

对 mem_map 查找一遍，并统计所保留的页面数。

```
             #ifdef CONFIG_HIGHMEM
             for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {
                  struct page *page = mem_map + tmp;
```

```
                if (!page_is_ram(tmp)) {
                    SetPageReserved(page);
                    continue;
                }
            if (bad_ppro && page_kills_ppro(tmp)) {
                SetPageReserved(page);
                continue;
            }
            ClearPageReserved(page);
            set_bit(PG_highmem, &page->flags);
            atomic_set(&page->count, 1);
            __free_page(page);
             totalhigh_pages++;
        }

        totalram_pages += totalhigh_pages;
        #endif
```

把高区内存查找一遍，并把保留但不能使用的页面标记为 PG_highmem，并调用__free_page()释放它，还要修改伙伴系统的位图。

```
        codesize =  (unsigned long) &_etext - (unsigned long) &_text;
        datasize =  (unsigned long) &_edata - (unsigned long) &_etext;
        initsize =  (unsigned long) &__init_end -
                                  (unsigned long) &__init_begin;
        printk("Memory: %luk/%luk available (%dk kernel code,
                    %dk reserved,
                    %dk data, %dk init, %ldk highmem)\n",
                     (unsigned long) nr_free_pages() <<
                                                (PAGE_SHIFT-10),
                    max_mapnr << (PAGE_SHIFT-10),
                    codesize >> 10,
                    reservedpages << (PAGE_SHIFT-10),
                    datasize >> 10,
                    initsize >> 10,
                     (unsigned long) (totalhigh_pages
                                      << (PAGE_SHIFT-10)));
```

计算内核各个部分的大小，并打印统计信息。

从以上的介绍可以看出，在初始化阶段，对内存的初始化要做许多工作。但这里要说明的是，尽管在这个阶段建立起了初步的虚拟内存管理机制，但仅仅考虑了内核虚拟空间（3GB以上），还根本没有涉及用户空间的管理。因此，在这个阶段，虚拟存储空间到物理存储空间的映射非常简单，仅仅通过一种简单的线性关系就可以达到虚地址到物理地址之间的相互转换。但是，了解这个初始化阶段又非常重要，它是后面进一步进行内存管理分析的基础。

6.3　内存的分配和回收

在内存初始化完成以后，内存中就常驻有内核映像（内核代码和数据）。以后，随着用户程序的执行和结束，就需要不断地分配和释放物理页面。内核应该为分配一组连续的页面

而建立一种稳定、高效的分配策略。为此，必须解决一个比较重要的内存管理问题，即外碎片问题。频繁地请求和释放不同大小的一组连续页面，必然导致在已分配的内存块中分散许多小块的空闲页面。由此带来的问题是，即使这些小块的空闲页面加起来足以满足所请求的页面，但是要分配一个大块的连续页面可能就根本无法满足。Linux 采用著名的伙伴（Buddy）系统算法来解决外碎片问题。

但是请注意，在 Linux 中，CPU 不能按物理地址来访问存储空间，而必须使用虚拟地址；因此，对于内存页面的管理，通常是先在虚存空间中分配一个虚存区间，然后才根据需要为此区间分配相应的物理页面并建立起映射，也就是说，虚存区间的分配在前，而物理页面的分配在后，但是为了承接上一节的内容，我们先介绍内存的分配和回收，然后再介绍用户进程虚存区间的建立。

6.3.1　伙伴算法

1．原理

Linux 的伙伴算法把所有的空闲页面分为 10 个块组，每组中块的大小是 2 的幂次方个页面，例如，第 0 组中块的大小都为 2^0（1 个页面），第 1 组中块的大小都为 2^1（2 个页面），第 9 组中块的大小都为 2^9（512 个页面）。也就是说，每一组中块的大小是相同的，且这同样大小的块形成一个链表。

我们通过一个简单的例子来说明该算法的工作原理。

假设要求分配的块的大小为 128 个页面（由多个页面组成的块我们就叫做页面块）。该算法先在块大小为 128 个页面的链表中查找，看是否有这样一个空闲块。如果有，就直接分配；如果没有，该算法会查找下一个更大的块，具体地说，就是在块大小 256 个页面的链表中查找一个空闲块。如果存在这样的空闲块，内核就把这 256 个页面分为两等份，一份分配出去，另一份插入到块大小为 128 个页面的链表中。如果在块大小为 256 个页面的链表中也没有找到空闲页块，就继续找更大的块，即 512 个页面的块。如果存在这样的块，内核就从 512 个页面的块中分出 128 个页面满足请求，然后从 384 个页面中取出 256 个页面插入到块大小为 256 个页面的链表中。然后把剩余的 128 个页面插入到块大小为 128 个页面的链表中。如果 512 个页面的链表中还没有空闲块，该算法就放弃分配，并发出出错信号。

以上过程的逆过程就是块的释放过程，这也是该算法名字的来由。满足以下条件的两个块称为伙伴：

（1）两个块的大小相同；

（2）两个块的物理地址连续。

伙伴算法把满足以上条件的两个块合并为一个块，该算法是迭代算法，如果合并后的块还可以跟相邻的块进行合并，那么该算法就继续合并。

2．数据结构

在 6.2.6 节中所介绍的管理区数据结构 struct zone_struct 中，涉及到空闲区数据结构：

free_area_t free_area[MAX_ORDER];

我们再次对 free_area_t 给予较详细的描述。

```
#difine   MAX_ORDER  10
    type struct free_area_struct {
           struct list_head   free_list
             unsigned  int    *map
     } free_area_t
```

其中 list_head 域是一个通用的双向链表结构，链表中元素的类型将为 mem_map_t（即 struct page 结构）。Map 域指向一个位图，其大小取决于现有的页面数。free_area 第 *k* 项位图的每一位，描述的就是大小为 2^k 个页面的两个伙伴块的状态。如果位图的某位为 0，表示一对兄弟块中或者两个都空闲，或者两个都被分配，如果为 1，肯定有一块已被分配。当兄弟块都空闲时，内核把它们当作一个大小为 2^{k+1} 的单独快来处理。如图 6.9 给出该数据结构的示意图。

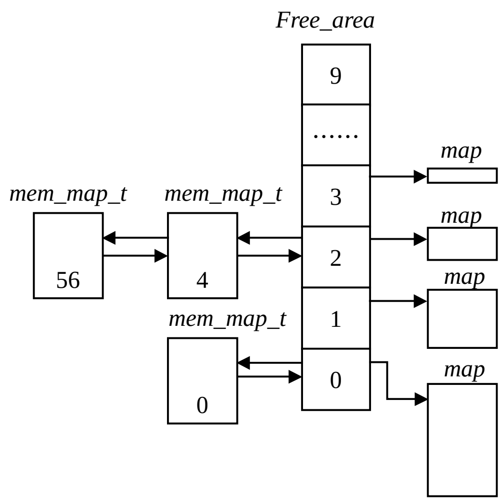


图 6.9　伙伴系统使用的数据结构

图 6.9 中，free_aea 数组的元素 0 包含了一个空闲页（页面编号为 0）；而元素 2 则包含了两个以 4 个页面为大小的空闲页面块，第一个页面块的起始编号为 4，而第二个页面块的起始编号为 56。

我们曾提到，当需要分配若干个内存页面时，用于 DMA 的内存页面必须是连续的。其实为了便于管理，从伙伴算法可以看出，只要请求分配的块大小不超过 512 个页面（2KB），内核就尽量分配连续的页面。

6.3.2　物理页面的分配和释放

当一个进程请求分配连续的物理页面时,可以通过调用 alloc_pages()来完成。Linux 2.4 版本中有两个 alloc_pages()，一个在 mm/numa.c 中，另一个在 mm/page_alloc,c 中，编译时根据所定义的条件选项 CONFIG_DISCONTIGMEM 来进行取舍。

1．非一致存储结构（NUMA）中页面的分配

CONFIG_DISCONTIGMEM 条件编译的含义是“不连续的存储空间”，Linux 把不连续的存储空间也归类为非一致存储结构（NUMA)。这是因为，不连续的存储空间本质上是一种广义的 NUMA，因为那说明在最低物理地址和最高物理地址之间存在着空洞，而有空洞的空间当然是“不一致”的。所以，在地址不连续的物理空间也要像结构不一样的物理空间那样划分出若干连续且均匀的“节点”。因此，在存储结构不连续的系统中，每个模块都有若干个节点，因而都有个 pg_data_t 数据结构队列。我们先来看 mm/numa.c 中的 alloc_page()函数：

```
/*
  * This can be refined. Currently, tries to do round robin, instead
  * should do concentratic circle search, starting from current node.
  */
struct page * _alloc_pages (unsigned int gfp_mask, unsigned int order)
{
        struct page *ret = 0;
        pg_data_t *start, *temp;
#ifndef CONFIG_NUMA
        unsigned long flags;
        static pg_data_t *next = 0;
#endif

        if  (order >= MAX_ORDER)
               return NULL;
#ifdef CONFIG_NUMA
        temp = NODE_DATA (numa_node_id () ) ;
#else
        spin_lock_irqsave (&node_lock, flags) ;
        if  (!next)   next = pgdat_list;
        temp = next;
        next = next->node_next;
        spin_unlock_irqrestore (&node_lock, flags) ;
#endif
        start = temp;
        while  (temp)  {
               if  ( (ret = alloc_pages_pgdat (temp, gfp_mask, order) ) )
                      return (ret) ;
               temp = temp->node_next;
        }
        temp = pgdat_list;
        while  (temp != start)  {
               if  ( (ret = alloc_pages_pgdat (temp, gfp_mask, order) ) )
                      return (ret) ;
               temp = temp->node_next;
        }
        return (0) ;
  }
```

对该函数的说明如下。

该函数有两个参数。gfp_mask 表示采用哪种分配策略。参数 order 表示所需物理块的大小，可以是 1、2、3 直到 $2^{MAX_ORDER-1}$。

如果定义了 CONFIG_NUMA，也就是在 NUMA 结构的系统中，可以通过 NUMA_DATA（）宏找到 CPU 所在节点的 pg_data_t 数据结构队列，并存放在临时变量 temp 中。

如果在不连续的 UMA 结构中,则有个 pg_data_t 数据结构的队列 pgdat_list,pgdat_list 就是该队列的首部。因为队列一般都是临界资源，因此，在对该队列进行两个以上的操作时要加锁。

分配时轮流从各个节点开始，以求各节点负荷的平衡。函数中有两个循环，其形式基本相同，也就是，对节点队列基本进行两遍扫描，直至在某个节点内分配成功，则跳出循环，否则，则彻底失败，从而返回 0。对于每个节点，调用 alloc_pages_pgdat（）函数试图分配所需的页面。

2．一致存储结构(UMA)中页面的分配

连续空间 UMA 结构的 alloc_page()是在 include/linux/mm.h 中定义的：

```
   #ifndef CONFIG_DISCONTIGMEM
     static inline struct page * alloc_pages (unsigned int gfp_mask, unsigned int order)
{
        /*
         * Gets optimized away by the compiler.
         */
        if  (order >= MAX_ORDER)
                return NULL;
        return __alloc_pages (gfp_mask, order,
                contig_page_data.node_zonelists+ (gfp_mask & GFP_ZONEMASK) ) ;
    }
    #endif
```

从这个函数的定义可以看出，alloc_page()是_alloc_pages（）的封装函数，而_alloc_pages（）才是伙伴算法的核心。这个函数定义于 mm/page_alloc.c 中，我们先对此函数给予概要描述。

_alloc_pages（）在管理区链表 zonelist 中依次查找每个区，从中找到满足要求的区，然后用伙伴算法从这个区中分配给定大小（2^{order}个）的页面块。如果所有的区都没有足够的空闲页面，则调用 swapper 或 bdflush 内核线程，把脏页写到磁盘以释放一些页面。

在__alloc_pages()和虚拟内存（简称 VM）的代码之间有一些复杂的接口（后面会详细描述)。每个区都要对刚刚被映射到某个进程 VM 的页面进行跟踪，被映射的页面也许仅仅做了标记，而并没有真正地分配出去。因为根据虚拟存储的分配原理，对物理页面的分配要尽量推迟到不能再推迟为止，也就是说，当进程的代码或数据必须装入到内存时，才给它真正分配物理页面。

搞清楚页面分配的基本原则后，我们对其代码具体分析如下：

```
/*
     * This is the 'heart' of the zoned buddy allocator:
     */
struct page * __alloc_pages (unsigned int gfp_mask, unsigned int order, zonelist_t *zonelist)
{
         unsigned long min;
         zone_t **zone, * classzone;
         struct page * page;
```

```
int freed;

  zone = zonelist->zones;
 classzone = *zone;
  min = 1UL << order;
 for (;;) {
         zone_t *z = * (zone++) ;
         if (!z)
                 break;

        min += z->pages_low;
        if (z->free_pages > min) {
                page = rmqueue (z, order) ;
                if (page)
                        return page;
        }
}
```

这是对一个分配策略中所规定的所有页面管理区的循环。循环中依次考察各个区中空闲页面的总量，如果总量尚大于“最低水位线”与所请求页面数之和，就调用 rmqueue（）试图从该区中进行分配。如果分配成功，则返回一个 page 结构指针，指向页面块中第一个页面的起始地址。

```
classzone->need_balance = 1;
mb () ;
if (waitqueue_active (&kswapd_wait) )
        wake_up_interruptible (&kswapd_wait) ;
```

如果发现管理区中的空闲页面总量已经降到最低点，则把 zone_t 结构中需要重新平衡的标志（need_balance）置 1，而且如果内核线程 kswapd 在一个等待队列中睡眠，就唤醒它，让它收回一些页面以备使用（可以看出，need_balance 是和 kswapd 配合使用的）。

```
zone = zonelist->zones;
min = 1UL << order;
for (;;) {
        unsigned long local_min;
        zone_t *z = * (zone++) ;
        if (!z)
                break;

        local_min = z->pages_min;
        if (! (gfp_mask & __GFP_WAIT) )
                local_min >>= 2;
        min += local_min;
        if (z->free_pages > min) {
                page = rmqueue (z, order) ;
                if (page)
                        return page;
        }
}
```

如果给定分配策略中所有的页面管理区都分配失败，那只好把原来的“最低水位”再向下调(除以 4)，然后看是否满足要求(z->free_pages > min)，如果能满足要求，则调用 rmqueue

() 进行分配。

```
            /* here we're in the low on memory slow path */

    rebalance:
            if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
                    zone = zonelist->zones;
                    for (;;) {
                            zone_t *z = *(zone++);
                            if (!z)
                                    break;

                            page = rmqueue(z, order);
                            if (page)
                                    return page;
                    }
                    return NULL;
            }
```

如果分配还不成功，这时候就要看是哪类进程在请求分配内存页面。其中 PF_MEMALLOC 和 PF_MEMDIE 是进程的 task_struct 结构中 flags 域的值，对于正在分配页面的进程（如 kswapd 内核线程），则其 PF_MEMALLOC 的值为 1（一般进程的这个标志为 0），而对于使内存溢出而被杀死的进程，则其 PF_MEMDIE 为 1。不管哪种情况，都说明必须给该进程分配页面（想想为什么）。因此，继续进行分配。

```
            /* Atomic allocations - we can't balance anything */
            if (!(gfp_mask & __GFP_WAIT))
                    return NULL;
```

如果请求分配页面的进程不能等待，也不能被重新调度，只好在没有分配到页面的情况下“空手”返回。

```
            page = balance_classzone(classzone, gfp_mask, order, &freed);
            if (page)
                    return page;
```

如果经过几番努力，必须得到页面的进程（如 kswapd）还没有分配到页面，就要调用 balance_classzone () 函数把当前进程所占有的局部页面释放出来。如果释放成功，则返回一个 page 结构指针，指向页面块中第一个页面的起始地址。

```
            zone = zonelist->zones;
            min = 1UL << order;
            for (;;) {
                    zone_t *z = *(zone++);
                    if (!z)
                            break;

                    min += z->pages_min;
                    if (z->free_pages > min) {
                            page = rmqueue(z, order);
                            if (page)
                                    return page;
                    }
            }
```

继续进行分配。

```
        /* Don't let big-order allocations loop *
        if  (order > 3)
               return NULL;

       /* Yield for kswapd, and try again */
        current->policy |= SCHED_YIELD;
        __set_current_state (TASK_RUNNING) ;
       schedule () ;
        goto rebalance;
      }
```

在这个函数中，频繁调用了 rmqueue（）函数，下面我们具体来看一下这个函数内容。

(1) rmqueue（）函数

该函数试图从一个页面管理区分配若干连续的内存页面。这是最基本的分配操作，其具体代码如下：

```
static struct page * rmqueue (zone_t *zone, unsigned int order)
{
        free_area_t * area = zone->free_area + order;
        unsigned int curr_order = order;
        struct list_head *head, *curr;
        unsigned long flags;
        struct page *page;

       spin_lock_irqsave (&zone->lock, flags) ;
        do {
                head = &area->free_list;
                curr = memlist_next (head) ;

                if  (curr != head)  {
                        unsigned int index;

                        page = memlist_entry (curr, struct page, list) ;
                        if  (BAD_RANGE (zone,page) )
                                BUG () ;
                        memlist_del (curr) ;
                        index = page - zone->zone_mem_map;
                        if  (curr_order != MAX_ORDER-1)
                                MARK_USED (index, curr_order, area) ;
                        zone->free_pages -= 1UL << order;

                        page = expand (zone, page, index, order, curr_order, area) ;
                        spin_unlock_irqrestore (&zone->lock, flags) ;

                        set_page_count (page, 1) ;
                        if  (BAD_RANGE (zone,page) )
                                BUG () ;
                        if  (PageLRU (page) )
                                BUG () ;
                        if  (PageActive (page) )
                                BUG () ;
                        return page;
```

```
                }
                curr_order++;
                area++;
        } while  (curr_order < MAX_ORDER) ;
        spin_unlock_irqrestore (&zone->lock, flags) ;

        return NULL;
 }
```

对该函数的解释如下。

参数 zone 指向要分配页面的管理区，order 表示要求分配的页面数为 2^{order}。

do 循环从 free_area 数组的第 order 个元素开始，扫描每个元素中由 page 结构组成的双向循环空闲队列。如果找到合适的页块，就把它从队列中删除，删除的过程是不允许其他进程、其他处理器来打扰的。所以要用 spin_lock_irqsave（）将这个循环加上锁。

首先在恰好满足大小要求的队列里进行分配。其中 memlist_entry(curr, struct page, list)获得空闲块的第 1 个页面的地址，如果这个地址是个无效的地址，就陷入 BUG()。如果有效，memlist_del(curr)从队列中摘除分配出去的页面块。如果某个页面块被分配出去，就要在 frea_area 的位图中进行标记，这是通过调用 MARK_USED（）宏来完成的。

如果分配出去后还有剩余块，就通过 expand（）获得所分配的页块，而把剩余块链入适当的空闲队列中。

如果当前空闲队列没有空闲块，就从更大的空闲块队列中找。

（2）expand（）函数

该函数源代码如下。

```
    static inline struct page * expand  (zone_t *zone, struct page *page,
         unsigned long index, int low, int high, free_area_t * area)
  {
        unsigned long size = 1 << high;

        while  (high > low)  {
                if  (BAD_RANGE (zone,page) )
                        BUG () ;
                area--;
                high--;
                size >>= 1;
                memlist_add_head (& (page) ->list, & (area) ->free_list) ;
                MARK_USED (index, high, area) ;
                index += size;
                page += size;
        }
        if  (BAD_RANGE (zone,page) )
                BUG () ;
        return page;
  }
```

对该函数解释如下。

参数 zone 指向已分配页块所在的管理区；page 指向已分配的页块；index 是已分配的页面在 mem_map 中的下标； low 表示所需页面块大小为 2^{low}，而 high 表示从空闲队列中实际进行分配的页面块大小为 2^{high}；area 是 free_area_struct 结构，指向实际要分配的页块。

通过上面介绍可以知道，返回给请求者的块大小为 2^{low} 个页面，并把剩余的页面放入合适的空闲队列，且对伙伴系统的位图进行相应的修改。例如，假定我们需要一个 2 页面的块，但是，我们不得不从 order 为 3（8 个页面）的空闲队列中进行分配，又假定我们碰巧选择物理页面 800 作为该页面块的底部。在我们这个例子中，这几个参数值为：

```
page == mem_map+800
index == 800
   low == 1
   high == 3
      area == zone->free_area+high  （ 也就是 frea_area 数组中下标为 3 的元素）
```

首先把 size 初始化为分配块的页面数(例如，size = 1<<3 == 8)

while 循环进行循环查找。每次循环都把 size 减半。如果我们从空闲队列中分配的一个块与所要求的大小匹配，那么 low = high，就彻底从循环中跳出，返回所分配的页块。

如果分配到的物理块所在的空闲块大于所需块的大小（即 $2^{high}>2^{low}$），那就将该空闲块分为两半（即 area--;high--; size >>= 1），然后调用 memlist_add_head（）把刚分配出去的页面块又加入到低一档（物理块减半）的空闲队列中，准备从剩下的一半空闲块中重新进行分配，并调用 MARK_USED（）设置位图。

在上面的例子中，第 1 次循环，我们从页面 800 开始，把页面大小为 4（即 2^{high--}）的块其首地址插入到 frea_area[2]中的空闲队列；因为 low<high，又开始第 2 次循环，这次从页面 804 开始，把页面大小为 2 的块插入到 frea_area[1]中的空闲队列，此时，page＝806，high=low＝1，退出循环，我们给调用者返回从 806 页面开始的一个 2 页面块。

从这个例子可以看出，这是一种巧妙的分配算法。

3．释放页面

从上面的介绍可以看出，页面块的分配必然导致内存的碎片化，而页面块的释放则可以将页面块重新组合成大的页面块。页面的释放函数为__free_pages(page struct *page, unsigned long order)，该函数从给定的页面开始，释放的页面块大小为 2^{order}。原函数为:

```
void __free_pages（page struct *page, unsigned long order）
{
            if （!PageReserved（page） && put_page_testzero（page））
            __free_pages_ok（page, order）;
}
```

其中比较巧妙的部分就是调用 put_page_testzero()宏，该函数把页面的引用计数减 1，如果减 1 后引用计数为 0，则该函数返回 1。因此，如果调用者不是该页面的最后一个用户，那么，这个页面实际上就不会被释放。另外要说明的是不可释放保留页 PageReserved，这是通过 PageReserved（）宏进行检查的。

如果调用者是该页面的最后一个用户，则__free_pages() 再调用 __free_pages_ok()。__free_pages_ok（）才是对页面块进行释放的实际函数，该函数把释放的页面块链入空闲链表，并对伙伴系统的位图进行管理，必要时合并伙伴块。这实际上是 expand()函数的反操作，我们对此不再进行详细的讨论。

6.3.3 Slab 分配机制

采用伙伴算法分配内存时，每次至少分配一个页面。但当请求分配的内存大小为几十个字节或几百个字节时应该如何处理？如何在一个页面中分配小的内存区，小内存区的分配所产生的内碎片又如何解决？

Linux 2.0 采用的解决办法是建立了 13 个空闲区链表，它们的大小从 32 字节到 132056 字节。从 Linux 2.2 开始，MM 的开发者采用了一种叫做 Slab 的分配模式，该模式早在 1994 年就被开发出来，用于 Sun Microsystem Solaris 2.4 操作系统中。Slab 的提出主要是基于以下考虑。

内核对内存区的分配取决于所存放数据的类型。例如，当给用户态进程分配页面时，内核调用 get_free_page()函数，并用 0 填充这个页面。而给内核的数据结构分配页面时，事情没有这么简单，例如，要对数据结构所在的内存进行初始化、在不用时要收回它们所占用的内存。因此，Slab 中引入了对象这个概念，所谓对象就是存放一组数据结构的内存区，其方法就是构造或析构函数，构造函数用于初始化数据结构所在的内存区，而析构函数收回相应的内存区。但为了便于理解，你也可以把对象直接看作内核的数据结构。为了避免重复初始化对象，Slab 分配模式并不丢弃已分配的对象，而是释放但把它们依然保留在内存中。当以后又要请求分配同一对象时，就可以从内存获取而不用进行初始化，这是在 Solaris 中引入 Slab 的基本思想。

实际上，Linux 中对 Slab 分配模式有所改进，它对内存区的处理并不需要进行初始化或回收。出于效率的考虑，Linux 并不调用对象的构造或析构函数，而是把指向这两个函数的指针都置为空。Linux 中引入 Slab 的主要目的是为了减少对伙伴算法的调用次数。

实际上，内核经常反复使用某一内存区。例如，只要内核创建一个新的进程，就要为该进程相关的数据结构（task_struct、打开文件对象等）分配内存区。当进程结束时，收回这些内存区。因为进程的创建和撤销非常频繁，因此，Linux 的早期版本把大量的时间花费在反复分配或回收这些内存区上。从 Linux 2.2 开始，把那些频繁使用的页面保存在高速缓存中并重新使用。

可以根据对内存区的使用频率来对它分类。对于预期频繁使用的内存区，可以创建一组特定大小的专用缓冲区进行处理，以避免内碎片的产生。对于较少使用的内存区，可以创建一组通用缓冲区（如 Linux 2.0 中所使用的 2 的幂次方）来处理，即使这种处理模式产生碎片，也对整个系统的性能影响不大。

硬件高速缓存的使用，又为尽量减少对伙伴算法的调用提供了另一个理由，因为对伙伴算法的每次调用都会“弄脏”硬件高速缓存，因此，这就增加了对内存的平均访问次数。

Slab 分配模式把对象分组放进缓冲区（尽管英文中使用了 Cache 这个词，但实际上指的是内存中的区域，而不是指硬件高速缓存）。因为缓冲区的组织和管理与硬件高速缓存的命中率密切相关，因此，Slab 缓冲区并非由各个对象直接构成，而是由一连串的“大块（Slab）”构成，而每个大块中则包含了若干个同种类型的对象，这些对象或已被分配，或空闲，如图 6.10 所示。一般而言，对象分两种，一种是大对象，一种是小对象。所谓小对象，是指在一

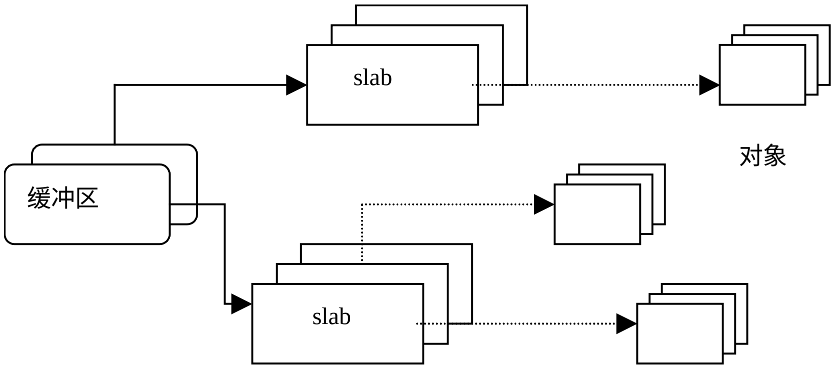


图 6.10　Slab 的组成

个页面中可以容纳下好几个对象的那种。例如，一个 inode 结构大约占 300 多个字节，因此，一个页面中可以容纳 8 个以上的 inode 结构，因此，inode 结构就为小对象。Linux 内核中把小于 512 字节的对象叫做小对象。

实际上，缓冲区就是主存中的一片区域，把这片区域划分为多个块，每块就是一个 Slab，每个 Slab 由一个或多个页面组成，每个 Slab 中存放的就是对象。

因为 Slab 分配模式的实现比较复杂，我们不准备对其进行详细的分析，只对主要内容给予描述。

1．Slab 的数据结构

Slab 分配模式有两个主要的数据结构，一个是描述缓冲区的结构 kmem_cache_t，一个是描述 Slab 的结构 kmem_slab_t，下面对这两个结构给予简要讨论。

（1）Slab

Slab 是 Slab 管理模式中最基本的结构。它由一组连续的物理页面组成，对象就被顺序放在这些页面中。其数据结构在 mm/slab.c 中定义如下：

```
/*
* slab_t
*
* Manages the objs in a slab. Placed either at the beginning of mem allocated
* for a slab, or allocated from an general cache.
* Slabs are chained into three list: fully used, partial, fully free slabs.
*/
 typedef struct slab_s {
       struct list_head        list;
       unsigned long           colouroff;
       void                   *s_mem;         /* including colour offset */
       unsigned int            inuse;          /* num of objs active in slab */
       kmem_bufctl_t           free;
} slab_t;
```

这里的链表用来将前一个 Slab 和后一个 Slab 链接起来形成一个双向链表，colouroff 为该 Slab 上着色区的大小，指针 s_mem 指向对象区的起点，inuse 是 Slab 中所分配对象的

个数。最后，free 的值指明了空闲对象链中的第一个对象，kmem_bufctl_t 其实是一个整数。Slab 结构的示意图如图 6.11 所示。

对于小对象，就把 Slab 的描述结构 slab_t 放在该 Slab 中；对于大对象，则把 Slab 结构游离出来，集中存放。关于 Slab 中的着色区再给予具体描述。

每个 Slab 的首部都有一个小小的区域是不用的，称为“着色区（Coloring Area）”。着色区的大小使 Slab 中的每个对象的起始地址都按高速缓存中的“缓存行（Cache Line）”大小进行对齐（80386 的一级高速缓存行大小为 16 字节，Pentium 为 32 字节)。因为 Slab 是由 1 个页面或多个页面（最多为 32）组成，因此，每个 Slab 都是从一个页面边界开始的，它自然按高速缓存的缓冲行对齐。但是，Slab 中的对象大小不确定，设置着色区的目的就是将 Slab 中第一个对象的起始地址往后推到与缓冲行对齐的位置。因为一个缓冲区中有多个 Slab，因此，应该把每个缓冲区中的各个 Slab 着色区的大小尽量安排成不同的大小，这样可以使得在不同的 Slab 中，处于同一相对位置的对象，让它们在高速缓存中的起始地址相互错开，这样就可以改善高速缓存的存取效率。

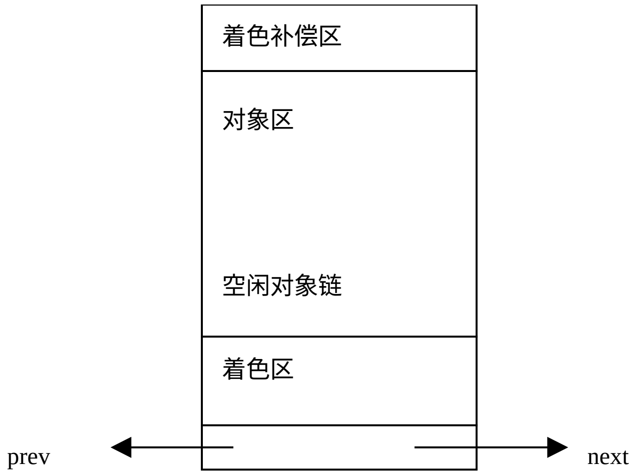


图 6.11　Slab 结构示意图

每个 Slab 上最后一个对象以后也有个小小的废料区是不用的，这是对着色区大小的补偿，其大小取决于着色区的大小，以及 Slab 与其每个对象的相对大小。但该区域与着色区的总和对于同一种对象的各个 Slab 是个常数。

每个对象的大小基本上是所需数据结构的大小。只有当数据结构的大小不与高速缓存中的缓冲行对齐时，才增加若干字节使其对齐。所以，一个 Slab 上的所有对象的起始地址都必然是按高速缓存中的缓冲行对齐的。

（2）缓冲区

每个缓冲区管理着一个 Slab 链表，Slab 按序分为 3 组。第 1 组是全满的 Slab（没有空闲的对象)，第 2 组 Slab 中只有部分对象被分配，部分对象还空闲，最后一组 Slab 中的对象全部空闲。只所以这样分组，是为了对 Slab 进行有效的管理。每个缓冲区还有一个轮转锁(Spinlock)，在对链表进行修改时用这个轮转锁进行同步。类型 kmem_cache_s 在 mm/slab.c

中定义如下：

```
 struct kmem_cache_s {
/* 1)  each alloc & free */
         /* full, partial first, then free */
        struct list_head        slabs_full;
         struct list_head        slabs_partial;
         struct list_head        slabs_free;
         unsigned int            objsize;
         unsigned int            flags;  /* constant flags */
         unsigned int            num;    /* # of objs per slab */
         spinlock_t              spinlock;
#ifdef CONFIG_SMP
         unsigned int            batchcount;
#endif

/* 2)  slab additions /removals */
         /* order of pgs per slab  (2^n)  */
         unsigned int            gfporder;

         /* force GFP flags, e.g. GFP_DMA */
         unsigned int            gfpflags;

         size_t                  colour;         /* cache colouring range */
         unsigned int            colour_off;     /* colour offset */
         unsigned int            colour_next;    /* cache colouring */
         kmem_cache_t            *slabp_cache;
         unsigned int            growing;
         unsigned int            dflags;         /* dynamic flags */

         /* constructor func */
         void  (*ctor)  (void *, kmem_cache_t *, unsigned long) ;

         /* de-constructor func */
         void  (*dtor)  (void *, kmem_cache_t *, unsigned long) ;

         unsigned long           failures;

/* 3)  cache creation/removal */
         char                    name[CACHE_NAMELEN];
         struct list_head        next;
#ifdef CONFIG_SMP
/* 4)  per-cpu data */
         cpucache_t              *cpudata[NR_CPUS];
#endif
…..
};
```

然后定义了 kmem_cache_t，并给部分域赋予了初值：

```
static kmem_cache_t cache_cache = {
         slabs_full:     LIST_HEAD_INIT (cache_cache.slabs_full) ,
         slabs_partial:  LIST_HEAD_INIT (cache_cache.slabs_partial) ,
         slabs_free:     LIST_HEAD_INIT (cache_cache.slabs_free) ,
```

```
        objsize:        sizeof (kmem_cache_t) ,
        flags:          SLAB_NO_REAP,
        spinlock:       SPIN_LOCK_UNLOCKED,
        colour_off:     L1_CACHE_BYTES,
        name:           "kmem_cache",
};
```

对该结构说明如下。

该结构中有 3 个队列 slabs_full、slabs_partial 以及 slabs_free，分别指向满 Slab、半满 Slab 和空闲 Slab，另一个队列 next 则把所有的专用缓冲区链成一个链表。

除了这些队列和指针外，该结构中还有一些重要的域：objsize 是原始的数据结构的大小，这里初始化为 kmem_cache_t 的大小；num 表示每个 Slab 上有几个缓冲区；gfporder 则表示每个 Slab 大小的对数，即每个 Slab 由 $2^{gfporder}$ 个页面构成。

如前所述，着色区的使用是为了使同一缓冲区中不同 Slab 上的对象区的起始地址相互错开，这样有利于改善高速缓存的效率。colour_off 表示颜色的偏移量，colour 表示颜色的数量；一个缓冲区中颜色的数量取决于 Slab 中对象的个数、剩余空间以及高速缓存行的大小。所以，对每个缓冲区都要计算它的颜色数量，这个数量就保存在 colour 中，而下一个 Slab 将要使用的颜色则保存在 colour_next 中。当 colour_next 达到最大值时，就又从 0 开始。着色区的大小可以根据（colour_off×colour）算得。例如，如果 colour 为 5，colour_off 为 8，则第一个 Slab 的颜色将为 0，Slab 中第一个对象区的起始地址（相对）为 0，下一个 Slab 中第一个对象区的起始地址为 8，再下一个为 16，24，32，0……等。

cache_cache 变量实际上就是缓冲区结构的头指针。

由此可以看出，缓冲区结构 kmem_cache_t 相当于 Slab 的总控结构，缓冲区结构与 Slab 结构之间的关系如图 6.12 所示。

在图 6.12 中，深灰色表示全满的 Slab，浅灰色表示含有空闲对象的 Slab，而无色表示空的 Slab。缓冲区结构之间形成一个单向链表，Slab 结构之间形成一个双向链表。另外，缓冲区结构还有分别指向满、半满、空闲 Slab 结构的指针。

2．专用缓冲区的建立和撤销

专用缓冲区是通过 kmem_cache_create（）函数建立的，函数原型为：

```
kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t offset,
        unsigned long c_flags,
        void (*ctor) (void *objp, kmem_cache_t *cachep, unsigned long flags) ,
        void (*dtor) (void *objp, kmem_cache_t *cachep, unsigned long flags) )
```

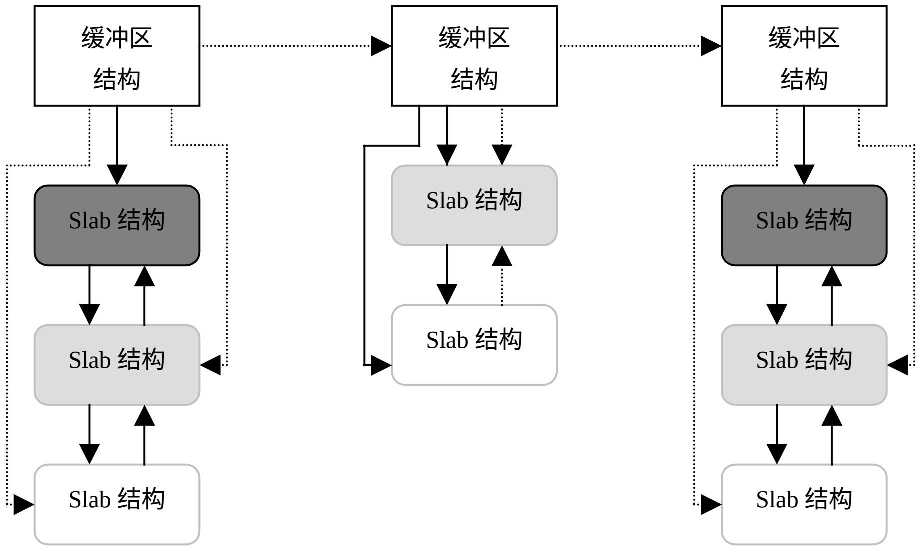


图 6.12 缓冲区结构 kmem_cache_t 与 Slab 结构 slab_t 之间的关系

对其参数说明如下。

（1）name： 缓冲区名 （ 19 个字符)。

（2）size： 对象大小。

（3）offset ：所请求的着色偏移量。

（4）c_flags ：对缓冲区的设置标志。

- SLAB_HWCACHE_ALIGN：表示与第一个高速缓存中的缓冲行边界（16 或 32 字节）对齐。
- SLAB_NO_REAP：不允许系统回收内存。
- SLAB_CACHE_DMA：表示 Slab 使用的是 DMA 内存。

（5）ctor ：构造函数（一般都为 NULL)。

（6）dtor ：析构函数（一般都为 NULL)。

（7）objp ：指向对象的指针。

（8）cachep ：指向缓冲区。

对专用缓冲区的创建过程简述如下。

kmem_cache_create（）函数要进行一系列的计算，以确定最佳的 Slab 构成。包括：每个 Slab 由几个页面组成，划分为多少个对象；Slab 的描述结构 slab_t 应该放在 Slab 的外面还是放在 Slab 的尾部；还有“颜色”的数量等等。并根据调用参数和计算结果设置 kmem_cache_t 结构中的各个域，包括两个函数指针 ctor 和 dtor。最后，将 kmem_cache_t 结构插入到 cache_cache 的 next 队列中。

但请注意，函数 kmem_cache_create（）所创建的缓冲区中还没有包含任何 Slab，因此，也没有空闲的对象。只有以下两个条件都为真时，才给缓冲区分配 Slab：

（1）已发出一个分配新对象的请求；

（2）缓冲区不包含任何空闲对象。

当这两个条件都成立时，Slab 分配模式就调用 kmem_cache_grow（）函数给缓冲区分配一个新的 Slab。其中，该函数调用 kmem_gatepages()从伙伴系统获得一组页面；然后又调用

kmem_cache_slabgmt()获得一个新的Slab结构；还要调用kmem_cache_init_objs()为新Slab中的所有对象申请构造方法（如果定义的话）；最后，调用 kmem_slab_link_end()把这个 Slab 结构插入到缓冲区中 Slab 链表的末尾。

Slab 分配模式的最大好处就是给频繁使用的数据结构建立专用缓冲区。但到目前的版本为止，Linux 内核中多数专用缓冲区的建立都用 NULL 作为构造函数的指针，例如，为虚存区间结构 vm_area_struct 建立的专用缓冲区 vm_area_cachep：

```
 vm_area_cachep = kmem_cache_create ("vm_area_struct",
                          sizeof (struct vm_area_struct) , 0,
                          SLAB_HWCACHE_ALIGN, NULL, NULL) ;
```

就把构造和析构函数的指针置为 NULL，也就是说，内核并没有充分利用 Slab 管理机制所提供的好处。为了说明如何利用专用缓冲区，我们从内核代码中选取一个构造函数不为空的简单例子，这个例子与网络子系统有关，在 net/core/buff.c 中定义：

```
  void __init  skb_init (void)
{
   int  i;
         skbuff_head_cache = kmem_cache_create ("skbuff_head_cache",
                                              sizeof (struct sk_buff) ,
                                               0,
                                               SLAB_HWCACHE_ALIGN,
                                               skb_headerinit, NULL) ;
        if  (!skbuff_head_cache)
              panic ("cannot create skbuff cache") ;

        for  (i=0; i<NR_CPUS; i++)
              skb_queue_head_init (&skb_head_pool[i].list) ;
     }
```

从代码中可以看出，skb_init () 调用 kmem_cache_create () 为网络子系统建立一个 sk_buff 数据结构的专用缓冲区，其名称为“ skbuff_head_cache ”(你可以通过读取／proc/slabinfo/文件得到所有缓冲区的名字)。调用参数 offset 为 0，表示第一个对象在 Slab 中的位移并无特殊要求。但是参数 flags 为 SLAB_HWCACHE_ALIGN，表示 Slab 中的对象要与高速缓存中的缓冲行边界对齐。对象的构造函数为 skb_headerinit ()，而析构函数为空，也就是说，在释放一个 Slab 时无需对各个缓冲区进行特殊的处理。

当从内核卸载一个模块时，同时应当撤销为这个模块中的数据结构所建立的缓冲区，这是通过调用 kmem_cache_destroy()函数来完成的。从 Linux 2.4.16 内核代码中进行查找可知，对这个函数的调用非常少。

3．通用缓冲区

在内核中初始化开销不大的数据结构可以合用一个通用的缓冲区。通用缓冲区非常类似于物理页面分配中的大小分区，最小的为 32，然后依次为 64、128、……直至 128KB（即 32 个页面），但是，对通用缓冲区的管理又采用的是 Slab 方式。从通用缓冲区中分配和释放缓冲区的函数为：

```
void *kmalloc (size_t size, int flags) ;
Void  kree (const void *objp) ;
```

因此，当一个数据结构的使用根本不频繁时，或其大小不足一个页面时，就没有必要给其分配专用缓冲区，而应该调用 kmallo()进行分配。如果数据结构的大小接近一个页面，则干脆通过 alloc_page()为之分配一个页面。

事实上，在内核中，尤其是驱动程序中，有大量的数据结构仅仅是一次性使用，而且所占内存只有几十个字节，因此，一般情况下调用 kmallo()给内核数据结构分配内存就足够了。另外，因为，在 Linux 2.0 以前的版本一般都调用 kmallo()给内核数据结构分配内存，因此，调用该函数的一个优点是（让你开发的驱动程序）能保持向后兼容。

6.3.4　内核空间非连续内存区的管理

我们说，任何时候，CPU 访问的都是虚拟内存，那么，在你编写驱动程序，或者编写模块时，Linux 给你分配什么样的内存？它处于 4GB 空间的什么位置？这就是我们要讨论的非连续内存。

首先，非连续内存处于 3GB 到 4GB 之间，也就是处于内核空间，如图 6.13 所示。

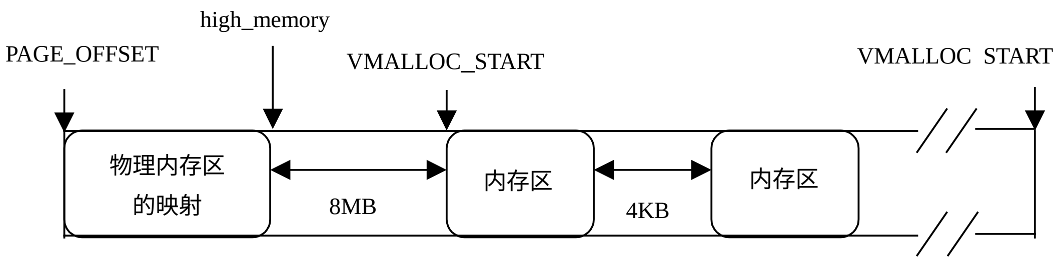


图 6.13　从 PAGE_OFFSET 开始的内核地址区间

图 6.13 中，PAGE_OFFSET 为 3GB，high_memory 为保存物理地址最高值的变量，VMALLOC_START 为非连续区的的起始地址，定义于 include／i386/pgtable.h 中：

```
        #define VMALLOC_OFFSET   (8*1024*1024)
        #define VMALLOC_START    ( ( (unsigned long)  high_memory + 2*VMALLOC_OFFSET-1)  & \~
(VMALLOC_OFFSET-1) )
```

在物理地址的末尾与第一个内存区之间插入了一个 8MB（VMALLOC_OFFSET）的区间，这是一个安全区，目的是为了“捕获”对非连续区的非法访问。出于同样的理由，在其他非连续的内存区之间也插入了 4KB 大小的安全区。每个非连续内存区的大小都是 4096 的倍数。

1．非连续区的数据结构

描述非连续区的数据结构为 struct vm_struct，定义于 include/linux/vmalloc.h 中：

```
struct vm_struct {
        unsigned long flags;
        void * addr;
        unsigned long size;
        struct vm_struct * next;
};
  struct vm_struct * vmlist;
```

非连续区组成一个单链表，链表第一个元素的地址存放在变量 vmlist 中。Addr 域是内存区的起始地址；size 是内存区的大小加 4096(安全区的大小)。

2．创建一个非连续区的结构

函数 get_vm_area（）创建一个新的非连续区结构，其代码在 mm/vmalloc.c 中：

```
struct vm_struct * get_vm_area (unsigned long size, unsigned long flags)
{
      unsigned long addr;
      struct vm_struct **p, *tmp, *area;

       area = (struct vm_struct *) kmalloc (sizeof (*area) , GFP_KERNEL) ;
       if (!area)
              return NULL;
       size += PAGE_SIZE;
       addr = VMALLOC_START;
       write_lock (&vmlist_lock) ;
      for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
              if ( (size + addr) < addr)
                     goto out;
              if (size + addr <= (unsigned long) tmp->addr)
                      break;
             addr = tmp->size + (unsigned long) tmp->addr;
              if (addr > VMALLOC_END-size)
                     goto out;
       }
       area->flags = flags;
       area->addr = (void *) addr;
       area->size = size;
       area->next = *p;
       *p = area;
       write_unlock (&vmlist_lock) ;
       return area;

out:
      write_unlock (&vmlist_lock) ;
       kfree (area) ;
       return NULL;
}
```

这个函数比较简单，就是在单链表中插入一个元素。其中调用了 kmalloc()和 kfree()函数,分别用来为 vm_struct 结构分配内存和释放所分配的内存。

3．分配非连续内存区

vmalloc（）函数给内核分配一个非连续的内存区，在/include/linux/vmalloc.h 中定义如下：

```
static inline void * vmalloc (unsigned long size)
{
       return __vmalloc (size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL) ;
}
```

vmalloc（）最终调用的是__vmalloc（）函数，该函数的代码在 mm/vmalloc.c 中：

```
void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)
{
        void * addr;
        struct vm_struct *area;

        size = PAGE_ALIGN (size) ;
        if (!size || (size >> PAGE_SHIFT) > num_physpages) {
                BUG () ;
                return NULL;
        }
        area = get_vm_area (size, VM_ALLOC) ;
        if (!area)
                return NULL;
        addr = area->addr;
        if (vmalloc_area_pages (VMALLOC_VMADDR (addr) , size, gfp_mask, prot) ) {
                vfree (addr) ;
                return NULL;
        }
        return addr;
}
```

函数首先把 size 参数取整为页面大小（4096）的一个倍数，也就是按页的大小进行对齐，然后进行有效性检查，如果有大小合适的可用内存，就调用 get_vm_area（）获得一个内存区的结构。但真正的内存区还没有获得，函数 vmalloc_area_pages（）真正进行非连续内存区的分配：

```
inline int vmalloc_area_pages (unsigned long address, unsigned long size,
                               int gfp_mask, pgprot_t prot)
{
        pgd_t * dir;
        unsigned long end = address + size;
        int ret;

        dir = pgd_offset_k (address) ;
        spin_lock (&init_mm.page_table_lock) ;
do {
                pmd_t *pmd;

                pmd = pmd_alloc (&init_mm, dir, address) ;
                ret = -ENOMEM;
                if (!pmd)
                        break;

                ret = -ENOMEM;
                if (alloc_area_pmd (pmd, address, end - address, gfp_mask, prot) )
                        break;

                address = (address + PGDIR_SIZE) & PGDIR_MASK;
                dir++;

                ret = 0;
        } while (address && (address < end) ) ;
```

```
        spin_unlock (&init_mm.page_table_lock) ;
        return ret;
    }
```

该函数有两个主要的参数，address 表示内存区的起始地址，size 表示内存区的大小。内存区的末尾地址赋给了局部变量 end。其中还调用了几个主要的函数或宏。

（1）pgd_offset_k（）宏导出这个内存区起始地址在页目录中的目录项。

（2）pmd_alloc（）为新的内存区创建一个中间页目录。

（3）alloc_area_pmd（）为新的中间页目录分配所有相关的页表，并更新页的总目录；该函数调用 pte_alloc_kernel()函数来分配一个新的页表，之后再调用 alloc_area_pte（）为页表项分配具体的物理页面。

（4）从 vmalloc_area_pages（）函数可以看出，该函数实际建立起了非连续内存区到物理页面的映射。

4．kmalloc()与 vmalloc()的区别

kmalloc()与 vmalloc() 都是在内核代码中提供给其他子系统用来分配内存的函数，但二者有何区别？

从前面的介绍已经看出，这两个函数所分配的内存都处于内核空间，即从 3GB～4GB；但位置不同，kmalloc()分配的内存处于 3GB～high_memory 之间，而 vmalloc()分配的内存在 VMALLOC_START～4GB 之间，也就是非连续内存区。一般情况下在驱动程序中都是调用 kmalloc()来给数据结构分配内存，而 vmalloc()用在为活动的交换区分配数据结构，为某些 I/O 驱动程序分配缓冲区，或为模块分配空间，例如在 include/asm-i386/module.h 中定义了如下语句：

```
#define module_map (x)          vmalloc (x)
```

其含义就是把模块映射到非连续的内存区。

与 kmalloc()和 vmalloc()相对应，两个释放内存的函数为 kfree()和 vfree()。

6.4　地址映射机制

顾名思义地址映射就是建立几种存储媒介（内存，辅存，虚存）间的关联，完成地址间的相互转换，它既包括磁盘文件到虚拟内存的映射，也包括虚拟内存到物理内存的映射，如图 6.14 所示。本节主要讨论磁盘文件到虚拟内存的映射，虚拟内存到物理内存的映射实际上是请页机制完成的（请看 6.5 节）。

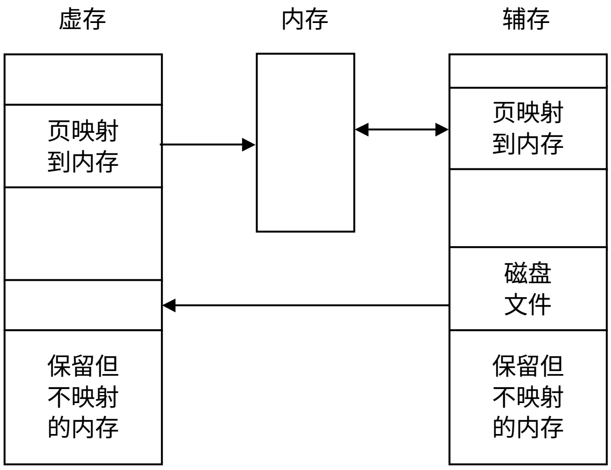


图 6.14　存储介质间的映射关系

6.4.1　描述虚拟空间的数据结构

前几节介绍的数据结构如存储节点（Node）、管理区（Zone）、页面（Page）及空闲区（free_area）都用于物理空间的管理。这一节主要关注虚拟空间的管理。虚拟空间的管理是以进程为基础的，每个进程都有各自的虚存空间（或叫用户空间，地址空间），除此之外，每个进程的“内核空间”是为所有的进程所共享的。

一个进程的虚拟地址空间主要由两个数据结构来描述。一个是最高层次的：mm_struct，一个是较高层次的：vm_area_structs。最高层次的 mm_struct 结构描述了一个进程的整个虚拟地址空间。较高层次的结构 vm_area_truct 描述了虚拟地址空间的一个区间(简称虚拟区)。

1．MM_STRUCT 结构

mm_strcut 用来描述一个进程的虚拟地址空间，在/include/linux/sched.h 中描述如下：

```
struct mm_struct {
        struct vm_area_struct * mmap;          /* 指向虚拟区间（VMA）链表 */
        rb_root_t mm_rb;                 ／*指向 red_black 树*/
        struct vm_area_struct * mmap_cache;      /* 指向最近找到的虚拟区间*/
        pgd_t * pgd;                     ／*指向进程的页目录*/
            atomic_t mm_users;                 /* 用户空间中的有多少用户*/
            atomic_t mm_count;               /* 对"struct mm_struct"有多少引用*/
        int map_count;                       /* 虚拟区间的个数*/
        struct rw_semaphore mmap_sem;
      spinlock_t page_table_lock;        /* 保护任务页表和 mm->rss */
        struct list_head mmlist;             /*所有活动（active）mm 的链表 */
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
```

```
            unsigned long def_flags;
            unsigned long cpu_vm_mask;
            unsigned long swap_address;

            unsigned dumpable:1;

            /* Architecture-specific MM context */
            mm_context_t context;
    };
```

对该结构进一步说明如下。

在内核代码中，指向这个数据结构的变量常常是 mm。

每个进程只有一个 mm_struct 结构，在每个进程的 task_struct 结构中，有一个指向该进程的结构。可以说，mm_struct 结构是对整个用户空间的描述。

一个进程的虚拟空间中可能有多个虚拟区间（参见下面对 vm_area_struct 描述），对这些虚拟区间的组织方式有两种，当虚拟区间较少时采用单链表，由 mmap 指针指向这个链表，当虚拟区间多时采用"红黑树（red_black tree）"结构，由 mm_rb 指向这颗树。在 2.4.10 以前的版本中，采用的是 AVL 树，因为与 AVL 树相比，对红黑树进行操作的效率更高。

因为程序中用到的地址常常具有局部性，因此，最近一次用到的虚拟区间很可能下一次还要用到，因此，把最近用到的虚拟区间结构应当放入高速缓存，这个虚拟区间就由 mmap_cache 指向。

指针 pgd 指向该进程的页目录（每个进程都有自己的页目录，注意同内核页目录的区别），当调度程序调度一个程序运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3）。

由于进程的虚拟空间及其下属的虚拟区间有可能在不同的上下文中受到访问，而这些访问又必须互斥，所以在该结构中设置了用于 P、V 操作的信号量 mmap_sem。此外，page_table_lock 也是为类似的目的而设置的。

虽然每个进程只有一个虚拟地址空间，但这个地址空间可以被别的进程来共享，如，子进程共享父进程的地址空间（也即共享 mm_struct 结构）。所以，用 mm_user 和 mm_count 进行计数。类型 atomic_t 实际上就是整数，但对这种整数的操作必须是"原子"的。

另外，还描述了代码段、数据段、堆栈段、参数段以及环境段的起始地址和结束地址。这里的段是对程序的逻辑划分，与我们前面所描述的段机制是不同的。

mm_context_t 是与平台相关的一个结构，对 i386 用处不大。

在后面对代码的分析中将对有些域给予进一步说明。

2. VM_AREA_STRUCT 结构

vm_area_struct 描述进程的一个虚拟地址区间，在/include/linux/mm.h 中描述如下：

```
struct vm_area_struct
        struct mm_struct * vm_mm;        /* 虚拟区间所在的地址空间*/
        unsigned long vm_start;          /* 在 vm_mm 中的起始地址*/
        unsigned long vm_end;            /*在 vm_mm 中的结束地址 */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;
```

```
        pgprot_t vm_page_prot;          /* 对这个虚拟区间的存取权限 */
        unsigned long vm_flags;         /* 虚拟区间的标志 */

        rb_node_t vm_rb;

        /*
         * For areas with an address space and backing store,
         * one of the address_space->i_mmap{,shared} lists,
         * for shm areas, the list of attaches, otherwise unused.
         */
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        /*对这个区间进行操作的函数 */
        struct vm_operations_struct * vm_ops;

        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        unsigned long vm_raend;         /* XXX: put full readahead info here. */
        void * vm_private_data;         /* was vm_pte (shared mem) */
};
```

vm_flag 是描述对虚拟区间的操作的标志，其定义和描述如表 6.1 所示。

表 6.1　虚拟区间的标志

标 志 名	描　　述
VM_DENYWRITE	在这个区间映射一个打开后不能用来写的文件
VM_EXEC	页可以被执行
VM_EXECUTABLE	页含有可执行代码
VM_GROWSDOWN	这个区间可以向低地址扩展
VM_GROWSUP	这个区间可以向高地址扩展
VM_IO	这个区间映射一个设备的 I/O 地址空间
VM_LOCKED	页被锁住不能被交换出去
VM_MAYEXEC	VM_EXEC 标志可以被设置
VM_MAYREAD	VM_READ 标志可以被设置
VM_MAYSHARE	VM_SHARE 标志可以被设置
VM_MAYWRITE	VM_WRITE 标志可以被设置
VM_READ	页是可读的
VM_SHARED	页可以被多个进程共享
VM_SHM	页用于 IPC 共享内存
VM_WRITE	页是可写的

较高层次的结构 vm_area_struct 是由双向链表连接起来的，它们是按虚地址的降顺序来排列的，每个这样的结构都对应描述一个相邻的地址空间范围。之所以这样分割，是因为

每个虚拟区间可能来源不同，有的可能来自可执行映像，有的可能来自共享库，而有的则可能是动态分配的内存区，所以对每一个由 vm_area_struct 结构所描述的区间的处理操作和它前后范围的处理操作不同。因此 Linux 把虚拟内存分割管理，并利用了虚拟内存处理例程（vm_ops）来抽象对不同来源虚拟内存的处理方法。不同的虚拟区间其处理操作可能不同，Linux 在这里利用了面向对象的思想，即把一个虚拟区间看成一个对象，用 vm_area_struct 描述了这个对象的属性，其中的 vm_operation s-stract 结构描述了在这个对象上的操作，其定义在／include／linux／mm.h 中：

```
   /*
   * These are the virtual MM functions - opening of an area, closing and
   * unmapping it  (needed to keep files on disk up-to-date etc) , pointer
   * to the functions called when a no-page or a wp-page exception occurs.
   */
struct vm_operations_struct {
        void  (*open)  (struct vm_area_struct * area) ;
        void  (*close)  (struct vm_area_struct * area) ;
struct page *  (*nopage)  (struct vm_area_struct * area, unsigned long address, int unused) ;
};
```

vm_operations 结构中包含的是函数指针；其中，open、close 分别用于虚拟区间的打开、关闭，而 nopage 用于当虚存页面不在物理内存而引起的“缺页异常”时所应该调用的函数。图 6.15 给出了虚拟区间的操作集。

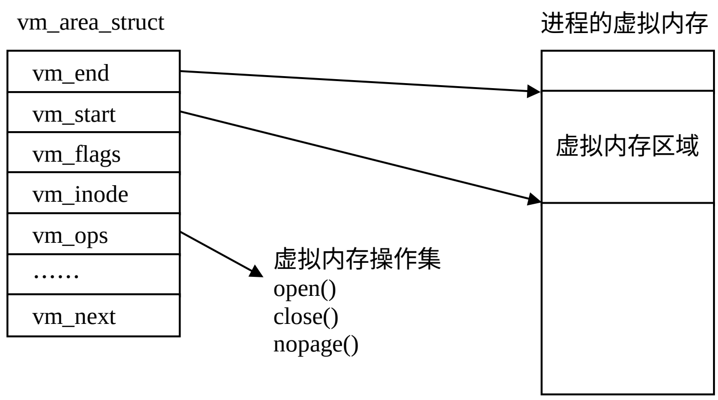


图 6.15　虚拟地址区间的操作集

3．红黑树结构

Linux 内核从 2.4.10 开始，对虚拟区间的组织不再采用 AVL 树，而是采用红黑树，这也是出于效率的考虑，虽然 AVL 树和红黑树很类似，但在插入和删除节点方面，采用红黑树的性能更好一些，下面对红黑树给予简单介绍。

一颗红黑树是具有以下特点的二叉树：

- 每个节点着有颜色，或者为红，或者为黑；
- 根节点为黑色；
- 如果一个节点为红色，那么它的子节点必须为黑色；

- 从一个节点到叶子节点上的所有路径都包含有相同的黑色节点数；

图 6.16 就是一颗红黑树。

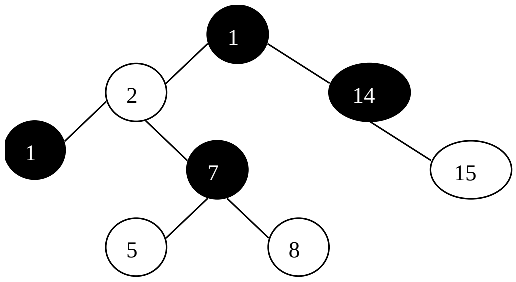


图 6.16　一颗红黑树

红黑树的结构在 include/linux/rbtree.h 中定义如下：

```
typedef struct rb_node_s
{
         struct rb_node_s * rb_parent;
         int rb_color;
#define RB_RED          0
#define RB_BLACK        1
        struct rb_node_s * rb_right;
        struct rb_node_s * rb_left;
} rb_node_t;
```

6.4.2　进程的虚拟空间

如前所述，每个进程拥有 3G 字节的用户虚存空间。但是，这并不意味着用户进程在这 3G 的范围内可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用。

那么，内核怎样管理每个进程 3GB 的虚存空间呢？概括地说，用户进程经过编译、链接后形成的映象文件有一个代码段和数据段（包括 data 段和 bss 段），其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，即全局变量和所有申明为 static 的局部变量，这些空间是进程所必需的基本要求，这些空间是在建立一个进程的运行映像时就分配好的。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的，如图 6.17 所示。

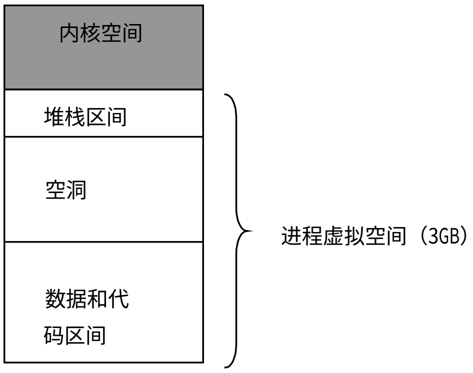


图 6.17　进程虚拟空间的划分

由图 6.17 可以看出，堆栈空间安排在虚存空间的顶部，运行时由顶向下延伸；代码段和数据段则在低部，运行时并不向上延伸。从数据段的顶部到堆栈段地址的下沿这个区间是一个巨大的空洞，这就是进程在运行时可以动态分配的空间（也叫动态内存）。

进程在运行过程中，可能会通过系统调用 mmap 动态申请虚拟内存或释放已分配的内存，新分配的虚拟内存必须和进程已有的虚拟地址链接起来才能使用；Linux 进程可以使用共享的程序库代码或数据，这样，共享库的代码和数据也需要链接到进程已有的虚拟地址中。在后面我们还会看到，系统利用了请页机制来避免对物理内存的过分使用。因为进程可能会访问当前不在物理内存中的虚拟内存，这时，操作系统通过请页机制把数据从磁盘装入到物理内存。为此，系统需要修改进程的页表，以便标志虚拟页已经装入到物理内存中，同时，Linux 还需要知道进程虚拟空间中任何一个虚拟地址区间的来源和当前所在位置，以便能够装入物理内存。

由于上面这些原因，Linux 采用了比较复杂的数据结构跟踪进程的虚拟地址。在进程的 task_struct 结构中包含一个指向 mm_struct 结构的指针。进程的 mm_struct 则包含装入的可执行映像信息以及进程的页目录指针 pgd。该结构还包含有指向 vm_area_struct 结构的几个指针，每个 vm_area_struct 代表进程的一个虚拟地址区间。

图 6.18 是某个进程的虚拟内存简化布局以及相应的几个数据结构之间的关系。从图中可

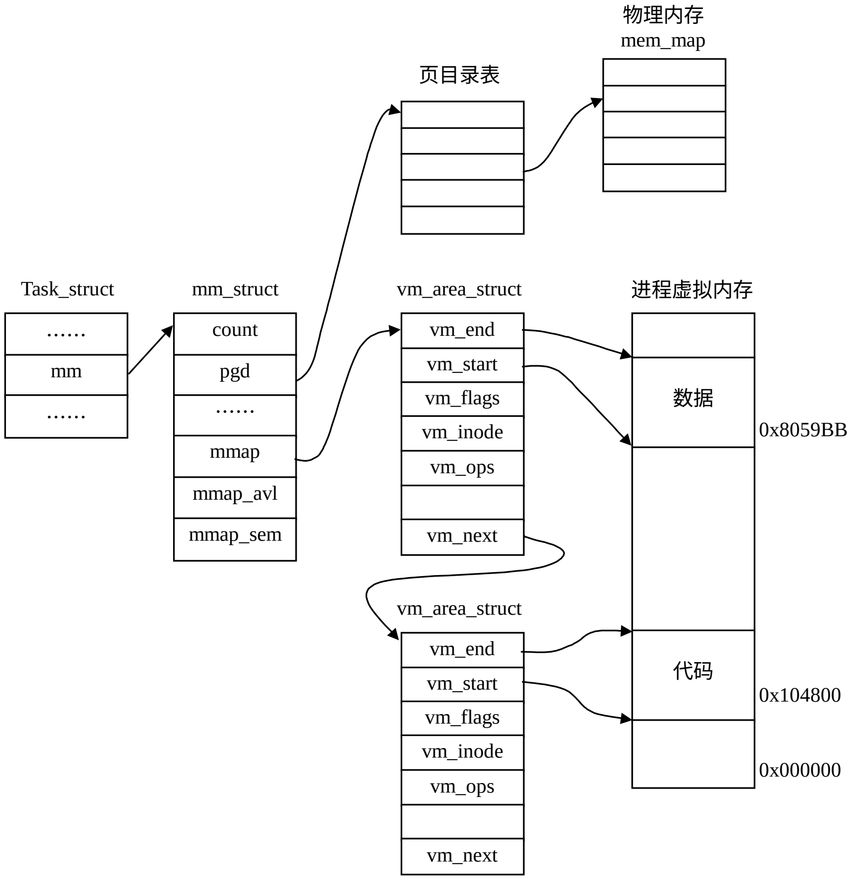


图 6.18　进程虚拟地址示意图

以看出，系统以虚拟内存地址的降序排列 vm_area_struct。在进程的运行过程中，Linux 要经常为进程分配虚拟地址区间，或者因为从交换文件中装入内存而修改虚拟地址信息，因此，vm_area_struct 结构的访问时间就成了性能的关键因素。为此，除链表结构外，Linux 还利用红黑（Red_black）树来组织 vm_area_struct。通过这种树结构，Linux 可以快速定位某个虚拟内存地址。

当进程利用系统调用动态分配内存时，Linux 首先分配一个 vm_area_struct 结构，并链接到进程的虚拟内存链表中，当后续的指令访问这一内存区间时，因为 Linux 尚未分配相应的物理内存,因此处理器在进行虚拟地址到物理地址的映射时会产生缺页异常(请看请页机制)，当 Linux 处理这一缺页异常时，就可以为新的虚拟内存区分配实际的物理内存。

在内核中，经常会用到这样的操作：给定一个属于某个进程的虚拟地址，要求找到其所属的区间以及 vma_area_struct 结构,这是由 find_vma()来实现的,其实现代码在 mm/mmap.c 中：

```
    * Look up the first VMA which satisfies  addr < vm_end,  NULL if none. */
struct vm_area_struct * find_vma (struct mm_struct * mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;
```

```
        if  (mm)  {
                /* Check the cache first. */
                /*  (Cache hit rate is typically around 35%.)  */
                vma = mm->mmap_cache;
                if  (! (vma && vma->vm_end > addr && vma->vm_start <= addr) )  {
                        rb_node_t * rb_node;

                        rb_node = mm->mm_rb.rb_node;
                        vma = NULL;

                        while  (rb_node)  {
                                struct vm_area_struct * vma_tmp;

                                vma_tmp = rb_entry (rb_node, struct vm_area_struct, vm_rb) ;

                                if  (vma_tmp->vm_end > addr)  {
                                        vma = vma_tmp;
                                        if  (vma_tmp->vm_start <= addr)
                                                break;
                                        rb_node = rb_node->rb_left;
                                } else
                                        rb_node = rb_node->rb_right;
                        }
                        if  (vma)
                                mm->mmap_cache = vma;
                }
        }
        return vma;
}
```

这个函数比较简单，我们对其主要点给予解释。

参数的含义：函数有两个参数，一个是指向 mm_struct 结构的指针，这表示一个进程的虚拟地址空间；一个是地址，表示该进程虚拟地址空间中的一个地址。

条件检查：首先检查这个地址是否恰好落在上一次（最近一次）所访问的区间中。根据代码作者的注释，命中率一般达到 35%，这也是 mm_struct 结构中设置 mmap_cache 指针的原因。如果没有命中，那就要在红黑树中进行搜索，红黑树与 AVL 树类似。

查找节点：如果已经建立了红黑树结构（rb_rode 不为空），就在红黑树中搜索。

如果找到指定地址所在的区间，就把 mmap_cache 指针设置成指向所找到的 vm_area_struct 结构。

如果没有找到，说明该地址所在的区间还没有建立，此时，就得建立一个新的虚拟区间，再调用 insert_vm_struct()函数将新建立的区间插入到 vm_struct 中的线性队列或红黑树中。

6.4.3 内存映射

当某个程序的映像开始执行时，可执行映像必须装入到进程的虚拟地址空间。如果该进程用到了任何一个共享库，则共享库也必须装入到进程的虚拟地址空间。由此可看出，Linux

并不将映像装入到物理内存，相反，可执行文件只是被连接到进程的虚拟地址空间中。随着程序的运行，被引用的程序部分会由操作系统装入到物理内存，这种将映像链接到进程地址空间的方法被称为“内存映射”。

当可执行映像映射到进程的虚拟地址空间时，将产生一组 vm_area_struct 结构来描述虚拟内存区间的起始点和终止点，每个 vm_area_struct 结构代表可执行映像的一部分，可能是可执行代码，也可能是初始化的变量或未初始化的数据，这些都是在函数 do_mmap()中来实现的。随着 vm_area_struct 结构的生成，这些结构所描述的虚拟内存区间上的标准操作函数也由 Linux 初始化。但要明确在这一步还没有建立从虚拟内存到物理内存的影射，也就是说还没有建立页表页目录。

为了对上面的原理进行具体的说明，我们来看一下 do_mmap()的实现机制。

函数 do_mmap()为当前进程创建并初始化一个新的虚拟区，如果分配成功，就把这个新的虚拟区与进程已有的其他虚拟区进行合并,do_mmap()在 include/linux/mm.h 中定义如下：

```
static inline unsigned long do_mmap (struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot,
        unsigned long flag, unsigned long offset)
{
        unsigned long ret = -EINVAL;
        if  ( (offset + PAGE_ALIGN (len) )  < offset)
                goto out;
        if  (! (offset & ~PAGE_MASK) )
             ret = do_mmap_pgoff (file, addr, len, prot, flag, offset >> PAGE_SHIFT) ;
out:
        return ret;
}
```

函数中参数的含义如下。

file：表示要映射的文件，file 结构将在第八章文件系统中进行介绍。

offset\：文件内的偏移量，因为我们并不是一下子全部映射一个文件，可能只是映射文件的一部分，off 就表示那部分的起始位置。

len：要映射的文件部分的长度。

addr：虚拟空间中的一个地址，表示从这个地址开始查找一个空闲的虚拟区。

prot: 这个参数指定对这个虚拟区所包含页的存取权限。可能的标志有 PROT_READ、PROT_WRITE、PROT_EXEC 和 PROT_NONE。前 3 个标志与标志 VM_READ、VM_WRITE 及 VM_EXEC 的意义一样。PROT_NONE 表示进程没有以上 3 个存取权限中的任意一个。

flag：这个参数指定虚拟区的其他标志：

MAP_GROWSDOWN，MAP_LOCKED，MAP_DENYWRITE 和 MAP_EXECUTABLE ：

它们的含义与表 6.1 中所列出标志的含义相同。

MAP_SHARED 和 MAP_PRIVATE ：

前一个标志指定虚拟区中的页可以被许多进程共享；后一个标志作用相反。这两个标志都涉及 vm_area_struct 中的 VM_SHARED 标志。

MAP_ANONYMOUS

表示这个虚拟区是匿名的，与任何文件无关。

MAP_FIXED

这个区间的起始地址必须是由参数 addr 所指定的。

MAP_NORESERVE
函数不必预先检查空闲页面的数目。

do_mmap()函数对参数 offset 的合法性检查后，就调用 do_mmap_pgoff（）函数，该函数才是内存映射的主要函数，do_mmap_pgoff（）的代码在 mm/mmap.c 中，代码比较长，我们分段来介绍：

```
unsigned long do_mmap_pgoff (struct file * file, unsigned long addr, unsigned long len,
        unsigned long prot, unsigned long flags, unsigned long pgoff)
{
        struct mm_struct * mm = current->mm;
        struct vm_area_struct * vma, * prev;
        unsigned int vm_flags;
        int correct_wcount = 0;
        int error;
        rb_node_t ** rb_link, * rb_parent;

        if  (file &&  (!file->f_op || !file->f_op->mmap) )
                return -ENODEV;

        if  ( (len = PAGE_ALIGN (len) )  == 0)
                return addr;

        if  (len > TASK_SIZE)
                return -EINVAL;

        /* offset overflow? */
        if  ( (pgoff +  (len >> PAGE_SHIFT) )  < pgoff)
                return -EINVAL;

        /* Too many mappings? */
        if  (mm->map_count > MAX_MAP_COUNT)
                return -ENOMEM;
```

函数首先检查参数的值是否正确，所提的请求是否能够被满足，如果发生以上情况中的任何一种，do_mmap()函数都终止并返回一个负值。

```
  /* Obtain the address to map to. we verify  (or select)  it and ensure
    * that it represents a valid section of the address space.
   */
        addr = get_unmapped_area (file, addr, len, pgoff, flags) ;
        if  (addr & ~PAGE_MASK)
                return addr;
```

调用 get_unmapped_area（）函数在当前进程的用户空间中获得一个未映射区间的起始地址。PAGE_MASK 的值为 0xFFFFF000，因此，如果“addr & ～PAGE_MASK”为非 0，说明 addr 最低 12 位非 0，addr 就不是一个有效的地址，就以这个地址作为返回值；否则，addr 就是一个有效的地址（最低 12 位为 0），继续向下看：

```
/* Do simple checking here so the lower-level routines won't have
* to. we assume access permissions have been handled by the open
* of the memory object, so we don't do any here.
*/
vm_flags = calc_vm_flags(prot,flags)  | mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
```

```
        /* mlock MCL_FUTURE? */
            if  (vm_flags & VM_LOCKED)  {
                    unsigned long locked = mm->locked_vm << PAGE_SHIFT;
                    locked += len;
                    if  (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
                            return -EAGAIN;
        }
```

如果 flag 参数指定的新虚拟区中的页必须锁在内存，且进程加锁页的总数超过了保存在进程的 task_struct 结构 rlim[RLIMIT_MEMLOCK].rlim_cur 域中的上限值，则返回一个负值。继续：

```
        if  (file)  {
                switch  (flags & MAP_TYPE)  {
                case MAP_SHARED:
                        if  ( (prot & PROT_WRITE)  && ! (file->f_mode & FMODE_WRITE) )
                                return -EACCES;

                        /* Make sure we don't allow writing to an append-only file.. */
    if  (IS_APPEND (file->f_dentry->d_inode)  &&  (file->f_mode &   FMODE_WRITE) )
                                return -EACCES;

                        /* make sure there are no mandatory locks on the file. */
                        if  (locks_verify_locked (file->f_dentry->d_inode) )
                                return -EAGAIN;

                        vm_flags |= VM_SHARED | VM_MAYSHARE;
                        if  (! (file->f_mode & FMODE_WRITE) )
                                vm_flags &= ~ (VM_MAYWRITE | VM_SHARED) ;

                        /* fall through */
                case MAP_PRIVATE:
                        if  (! (file->f_mode & FMODE_READ) )
                                return -EACCES;
                        break;

                default:
                        return -EINVAL;
                }
        } else {
                vm_flags |= VM_SHARED | VM_MAYSHARE;
                switch  (flags & MAP_TYPE)  {
                default:
                        return -EINVAL;
                case MAP_PRIVATE:
                        vm_flags &= ~ (VM_SHARED | VM_MAYSHARE) ;
                        /* fall through */
                case MAP_SHARED:
                        break;
                }
        }
```

如果 file 结构指针为 0，则目的仅在于创建虚拟区间，或者说，并没有真正的映射发生；如果 file 结构指针不为 0，则目的在于建立从文件到虚拟区间的映射，那就要根据标志指定的映射种类，把为文件设置的访问权考虑进去。

如果所请求的内存映射是共享可写的，就要检查要映射的文件是为写入而打开的，而不是以追加模式打开的，还要检查文件上没有上强制锁。

对于任何种类的内存映射，都要检查文件是否为读操作而打开的。

如果以上条件都不满足，就返回一个错误码。

```
/* Clear old maps */
        error = -ENOMEM;
munmap_back:
        vma = find_vma_prepare (mm, addr, &prev, &rb_link, &rb_parent) ;
        if  (vma && vma->vm_start < addr + len)  {
               if  (do_munmap (mm, addr, len) )
                     return -ENOMEM;
               goto munmap_back;
        }
```

函数 find_vma_prepare（）与 find_vma（）基本相同，它扫描当前进程地址空间的 vm_area_struct 结构所形成的红黑树，试图找到结束地址高于 addr 的第 1 个区间；如果找到了一个虚拟区，说明 addr 所在的虚拟区已经在使用，也就是已经有映射存在，因此要调用 do_munmap () 把这个老的虚拟区从进程地址空间中撤销，如果撤销不成功，就返回一个负数；如果撤销成功，就继续查找，直到在红黑树中找不到 addr 所在的虚拟区，并继续下面的检查：

```
    /* Check against address space limit. */
       if  ( (mm->total_vm << PAGE_SHIFT)  + len
          > current->rlim[RLIMIT_AS].rlim_cur)
              return -ENOMEM;
```

total_vm 是表示进程地址空间的页面数，如果把文件映射到进程地址空间后，其长度超过了保存在当前进程 rlim[RLIMIT_AS].rlim_cur 中的上限值，则返回一个负数。

```
  /* Private writable mapping? Check memory availability.. */
       if  ( (vm_flags &  (VM_SHARED | VM_WRITE) )  == VM_WRITE &&
          ! (flags & MAP_NORESERVE)  &&!vm_enough_memory (len >> PAGE_SHIFT) )
              return -ENOMEM;
```

如果 flags 参数中没有设置 MAP_NORESERVE 标志，新的虚拟区含有私有的可写页，空闲页面数小于要映射的虚拟区的大小；则函数终止并返回一个负数；其中函数 vm_enough_memory（）用来检查一个进程的地址空间中是否有足够的内存来进行一个新的映射。

```
        /* Can we just expand an old anonymous mapping? */
     if  (!file && ! (vm_flags & VM_SHARED)  && rb_parent)
            if  (vma_merge (mm, prev, rb_parent, addr, addr + len, vm_flags) )
                  goto out;
```

如果是匿名映射（file 为空），并且这个虚拟区是非共享的，则可以把这个虚拟区和与它紧挨的前一个虚拟区进行合并；虚拟区的合并是由 vma_merge()函数实现的。如果合并成功，则转 out 处，请看后面 out 处的代码。

```
   /* Determine the object being mapped and call the appropriate
      * specific mapper. the address has already been validated, but
      * not unmapped, but the maps are removed from the list.
```

```
        */
        vma = kmem_cache_alloc (vm_area_cachep, SLAB_KERNEL) ;
        if  (!vma)
                return -ENOMEM;
        vma->vm_mm = mm;
        vma->vm_start = addr;
        vma->vm_end = addr + len;
        vma->vm_flags = vm_flags;
        vma->vm_page_prot = protection_map[vm_flags & 0x0f];
        vma->vm_ops = NULL;
        vma->vm_pgoff = pgoff;
        vma->vm_file = NULL;
        vma->vm_private_data = NULL;
        vma->vm_raend = 0;
```

经过以上各种检查后，现在必须为新的虚拟区分配一个 vm_area_struct 结构。这是通过调用 Slab 分配函数 kmem_cache_alloc（）来实现的，然后就对这个结构的各个域进行了初始化。

```
            if  (file)  {
                error = -EINVAL;
                if  (vm_flags &  (VM_GROWSDOWN|VM_GROWSUP) )
                        goto free_vma;
                if  (vm_flags & VM_DENYWRITE)  {
                        error = deny_write_access (file) ;
                        if  (error)
                                goto free_vma;
                        correct_wcount = 1;
                }
                vma->vm_file = file;
                get_file (file) ;
                error = file->f_op->mmap (file, vma) ;
                if  (error)
                        goto unmap_and_free_vma;
        } else if  (flags & MAP_SHARED)  {
                error = shmem_zero_setup (vma) ;
                if  (error)
                        goto free_vma;
        }
free_vma:
        kmem_cache_free (vm_area_cachep, vma) ;
        return error;
}
```

如果建立的是从文件到虚存区间的映射，则情况下。

当参数 flags 中的 VM_GROWSDOWN 或 VM_GROWSUP 标志位为 1 时，说明这个区间可以向低地址或高地址扩展，但从文件映射的区间不能进行扩展，因此转到 free_vma，释放给 vm_area_struct 分配的 Slab，并返回一个错误。

当 flags 中的 VM_DENYWRITE 标志位为 1 时，就表示不允许通过常规的文件操作访问该文件，所以要调用 deny_write_access（）排斥常规的文件操作（参见第八章）。

get_file（）函数的主要作用是递增 file 结构中的共享计数。

每个文件系统都有个 fiel_operation 数据结构，其中的函数指针 mmap 提供了用来建立从该类文件到虚存区间进行映射的操作，这是最具有实质意义的函数；对于大部分文件系统，这个函数为 generic_file_mmap( )函数实现的，该函数执行以下操作。

初始化 vm_area_struct 结构中的 vm_ops 域。如果 VM_SHARED 标志为 1，就把该域设置成 file_shared_mmap，否则就把该域设置成 file_private_mmap。从某种意义上说，这个步骤所做的事情类似于打开一个文件并初始化文件对象的方法。

从索引节点的 i_mode 域（参见第八章）检查要映射的文件是否是一个常规文件。如果是其他类型的文件（例如目录或套接字），就返回一个错误代码。

从索引节点的 i_op 域中检查是否定义了 readpage( )的索引节点操作。如果没有定义，就返回一个错误代码。

调用 update_atime( )函数把当前时间存放在该文件索引节点的 i_atime 域中，并将这个索引节点标记成脏。

如果 flags 参数中的 MAP_SHARED 标志位为 1，则调用 shmem_zero_setup（）进行共享内存的映射。

继续看 do_mmap()中的代码。

```
/* Can addr have changed??
 *
 * Answer: Yes, several device drivers can do it in their
*         f_op->mmap method. -DaveM
 */
addr = vma->vm_start;
```

源码作者给出了解释，意思是说，addr 有可能已被驱动程序改变，因此，把新虚拟区的起始地址赋给 addr。

```
vma_link (mm, vma, prev, rb_link, rb_parent) ;
if  (correct_wcount)
        atomic_inc (&file->f_dentry->d_inode->i_writecount) ;
```

此时，应该把新建的虚拟区插入到进程的地址空间，这是由函数 vma_link（）完成的，该函数具有 3 方面的功能：

（1）把 vma 插入到虚拟区链表中；

（2）把 vma 插入到虚拟区形成的红黑树中；

（3）把 vam 插入到索引节点（inode）共享链表中。

函数 atomic_inc（x）给*x 加 1，这是一个原子操作。在内核代码中，有很多地方调用了以 atomic 为前缀的函数。所谓原子操作，就是在操作过程中不会被中断。

```
out:
        mm->total_vm += len >> PAGE_SHIFT;
        if  (vm_flags & VM_LOCKED)  {
                mm->locked_vm += len >> PAGE_SHIFT;
                make_pages_present (addr, addr + len) ;
        }
        return addr;
```

do_mmap()函数准备从这里退出，首先增加进程地址空间的长度，然后看一下对这个区间是否加锁，如果加锁，说明准备访问这个区间，就要调用 make_pages_present（）函数，建立虚拟页面到物理页面的映射，也就是完成文件到物理内存的真正调入。返回一个正数，

说明这次映射成功。

```
unmap_and_free_vma:
        if (correct_wcount)
               atomic_inc (&file->f_dentry->d_inode->i_writecount);
        vma->vm_file = NULL;
        fput (file);

       /* Undo any partial mapping done by a device driver. */
        zap_page_range (mm, vma->vm_start, vma->vm_end - vma->vm_start);
```

如果对文件的操作不成功，则解除对该虚拟区间的页面映射，这是由 zap_page_range() 函数完成的。

当你读到这里时可能感到困惑，页面的映射到底在何时建立？实际上，generic_file_mmap( )就是真正进行映射的函数。因为这个函数的实现涉及很多文件系统的内容，我们在此不进行深入的分析，当读者了解了文件系统的有关内容后，可自己进行分析。

这里要说明的是，文件到虚存的映射仅仅是建立了一种映射关系，也就是说，虚存页面到物理页面之间的映射还没有建立。当某个可执行映象映射到进程虚拟内存中并开始执行时，因为只有很少一部分虚拟内存区间装入到了物理内存，可能会遇到所访问的数据不在物理内存。这时，处理器将向 Linux 报告一个页故障及其对应的故障原因，于是就用到了请页机制。

6.5　请页机制

Linux 采用请页机制来节约内存，它仅仅把当前正在执行的程序要使用的虚拟页（少量一部分）装入内存。当需要访问尚未装入物理内存的虚拟内存区域时，处理器将向 Linux 报告一个页故障及其对应的故障原因。本节将主要介绍 arch／i386／mm／fault.c 中的页故障处理函数 do_page_fault，为了突出主题，我们将分析代码中的主要部分。

6.5.1　页故障的产生

页故障的产生有 3 种原因。

（1）一是程序出现错误，例如向随机物理内存中写入数据，或页错误发生在 TASK_SIZE（3G）的范围外，这些情况下，虚拟地址无效，Linux 将向进程发送 SIGSEGV 信号并终止进程的运行。

（2）另一种情况是，虚拟地址有效，但其所对应的页当前不在物理内存中，即缺页错误，这时，操作系统必须从磁盘映像或交换文件（此页被换出）中将其装入物理内存。这是本节要讨论的主要内容。

（3）最后一种情况是，要访问的虚地址被写保护，即保护错误，这时，操作系统必须判断：如果是用户进程正在写当前进程的地址空间，则发 SIGSEGV 信号并终止进程的运行；如果错误发生在一旧的共享页上时，则处理方法有所不同，也就是要对这一共享页进行复制，这就是我们后面要讲的写时复制（Copy On Write 简称 COW）技术。

有关页错误的发生次数的信息可在目录 proc/stat 下找到。

6.5.2 页错误的定位

页错误的定位既包含虚拟地址的定位，也包含被调入页在交换文件(swapfile)或在可执行映象中的定位。

具体地说，在一个进程访问一个无效页表项时，处理器产生一个陷入并报告一个页错误，它描述了页错误发生的虚地址和访问类型，这些类型通过页的错误码 error_code 中的前 3 位来判别 ，具体如下：

```
 *   bit 0 == 0 means no page found, 1 means protection fault
 *   bit 1 == 0 means read, 1 means write
 *   bit 2 == 0 means kernel, 1 means user-mode。
```

也就是说，如果第 0 位为 0，则错误是由访问一个不存在的页引起的（页表的表项中 present 标志为 0），否则，如果第 0 位为 1，则错误是由无效的访问权所引起的；如果第 1 位为 0，则错误是由读访问或执行访问所引起，如果为 1，则错误是由写访问所引起的；如果第 2 位为 0，则错误发生在处理器处于内核态时，否则，错误发生在处理器处于用户态时。

页错误的线性地址被存于 CR2 寄存器，操作系统必须在 vm_area_struct 中找到页错误发生时页的虚拟地址（通过红黑树或旧版本中的 AVL 树），下面通过 do_page_fault()中的一部分源代码来说明这个问题：

```
 /* CR2 中包含有最新的页错误发生时的虚拟地址*/
    __asm__ ("movl %%cr2,%0":"=r" (address) ) ;
        vma = find_vma (current, address) ;
```

如果没找到，则说明访问了非法虚地址，Linux 会发信号终止进程（如果必要）。否则，检查页错误类型，如果是非法类型(越界错误，段权限错误等）同样会发信号终止进程，部分源代码如下：

```
vma = find_vma (current, address) ;
     if  (!vma)
         goto bad_area;
   if  (vma->vm_start <= address)
         goto good_area;
   if  (! (vma->vm_flags & VM_GROWSDOWN) )
         goto bad_area;
     if  (error_code & 4)  {  /*如是用户态进程*/
         /* 不可访问堆栈空间*/
          if  (address + 32 < regs->esp)
              goto bad_area;
     }
     if  (expand_stack (vma, address) )
         goto bad_area;
bad_area:                  /* 用户态的访问*/
{
    if  (error_code & 4)  {
         current->tss.cr2 = address;
         current->tss.error_code = error_code;
         current->tss.trap_no = 14;
         force_sig (SIGSEGV, current) ;    /* 给当前进程发杀死信号*/
         return;
       ……
```

```
    die_if_kernel ("Oops", regs, error_code) ;        /*报告内核 */
     do_exit (SIGKILL) ;                          /*强行杀死进程*/
                      }
```

6.5.3　进程地址空间中的缺页异常处理

对有效的虚拟地址，如果是缺页错误，Linux 必须区分页所在的位置，即判断页是在交换文件中，还是在可执行映像中。为此，Linux 通过页表项中的信息区分页所在的位置。如果该页的页表项是无效的，但非空，则说明该页处于交换文件中，操作系统要从交换文件装入页。对于有效的虚拟地址 address，do_page_fault(　)转到 good_area 标号处的语句执行：

```
good_area:
write = 0;
if  (error_code & 2)  { /* 写访问 */
    if  (! (vma->vm_flags & VM_WRITE) )
        goto bad_area;
    write++;
} else                /* 读访问 */
    if  (error_code & 1 ||
        ! (vma->vm_flags &  (VM_READ | VM_EXEC) ) )
        goto bad_area;
```

如果错误由写访问引起，函数检查这个虚拟区是否可写。如果不可写，跳到 bad_area 代码处；如果可写，把 write 局部变量置为 1。

如果错误由读或执行访问引起，函数检查这一页是否已经存在于物理内存中。如果在，错误的发生就是由于进程试图访问用户态下的一个有特权的页面（页面的 User/Supervisor 标志被清除），因此函数跳到 bad_area 代码处（实际上这种情况从不发生，因为内核根本不会给用户进程分配有特权的页面）。如果不存在物理内存，函数还将检查这个虚拟区是否可读或可执行。

如果这个虚拟区的访问权限与引起错误的访问类型相匹配，则调用 handle_mm_fault(　) 函数：

```
if  (!handle_mm_fault (tsk, vma, address, write) )  {
    tsk->tss.cr2 = address;
    tsk->tss.error_code = error_code;
    tsk->tss.trap_no = 14;
    force_sig (SIGBUS, tsk) ;
    if  (! (error_code & 4) )  /* 内核态 */
        goto no_context;
}
```

如果 handle_mm_fault(　)函数成功地给进程分配一个页面，则返回 1；否则返回一个适当的错误码，以便 do_page_fault(　)函数可以给进程发送 SIGBUS 信号。handle_mm_fault(　)函数有 4 个参数：tsk 指向错误发生时正在 CPU 上运行的进程；vma 指向引起错误的虚拟地址所在虚拟区；address 为引起错误的虚拟地址；write：如果 tsk 试图向 address 写，则置为 1，如果 tsk 试图读或执行 address，则置为 0。

handle_mm_fault () 函数首先检查用来映射 address 的页中间目录和页表是否存在。即使 address 属于进程的地址空间，但相应的页表可能还没有分配，因此，在做别的事情之

前首先执行分配页目录和页表的任务：

```
pgd = pgd_offset (vma->vm_mm, address) ;
pmd = pmd_alloc (pgd, address) ;
if  (!pmd)
     return -1;
    pte = pte_alloc (pmd, address) ;
if  (!pte)
return -1;
```

pgd_offset()宏计算出 address 所在页在页目录中的目录项指针。如果有中间目录（i386 不起作用），调用 pmd_alloc(　)函数分配一个新的中间目录。然后，如果需要，调用 pte_alloc(　)函数分配一个新的页表。如果这两步都成功，pte 局部变量所指向的页表表项就是引用 address 的表项。然后调用 handle_pte_fault(　)函数检查 address 地址所对应的页表表项:

```
return handle_pte_fault (tsk, vma, address, write_access, pte) ;
```

handle_pte_fault(　)函数决定怎样给进程分配一个新的页面。

如果被访问的页不存在，也就是说，这个页还没有被存放在任何一个页面中，那么，内核分配一个新的页面并适当地初始化。这种技术称为请求调页。

如果被访问的页存在但是被标为只读，也就是说，它已经被存放在一个页面中，那么，内核分配一个新的页面，并把旧页面的数据拷贝到新页面来初始化它的内容。这种技术称为写时复制。

6.5.4　请求调页

请求调页指的是一种动态内存分配技术，它把页面的分配推迟到不能再推迟为止，也就是说，一直推迟到进程要访问的页不在物理内存时为止，由此引起一个缺页错误。

请求调页技术的引入主要是因为进程开始运行的时候并不访问其地址空间中的全部地址。事实上，有一部分地址也许进程永远不使用。此外，程序的局部性原理保证了在程序执行的每个阶段，真正使用的进程页只有一小部分，因此临时用不着的页所在的物理页面可以由其他进程来使用。因此，对于全局分配（一开始就给进程分配所需要的全部页面，直到程序结束才释放这些页面）来说，请求调页是首选的，因为它增加了系统中的空闲页面的平均数，从而更好地利用空闲内存。从另一个观点来看，在内存总数保持不变的情况下，请求调页从总体上能使系统有更大的吞吐量。

为这一切优点付出的代价是系统额外的开销：由请求调页所引发的每个“缺页”错误必须由内核处理，这将浪费 CPU 的周期。幸运的是，局部性原理保证了一旦进程开始在一组页上运行，在接下来相当长的一段时间内它会一直停留在这些页上而不去访问其他的页：这样我们就可以认为“缺页”错误是一种稀有事件。

基于以下原因，被寻址的页可以不在主存中。

(1) 进程永远也没有访问到这个页。内核能够识别这种情况，这是因为页表相应的表项被填充为 0，也就是说，pte_none 宏返回 1。

(2) 进程已经访问过这个页，但是这个页的内容被临时保存在磁盘上。内核能够识别这种情况，这是因为页表相应表项没被填充为 0（然而，由于页面不存在物理内存中，Present

为 0)。

handle_pte_fault（ ）函数通过检查与 address 相关的页表表项来区分这两种情况：

```
entry = *pte;
if (!pte_present (entry) ) {
    if (pte_none (entry) )
        return do_no_page (tsk, vma, address, write_access,
                           pte) ;
    return do_swap_page (tsk, vma, address, pte, entry,
                        write_access) ;
}
```

我们将在交换机制一节检查页被保存到磁盘上的这种情况(do_swap_page( ) 函数)。

在其他情况下，当页从未被访问时则调用 do_no_page( )函数。有两种方法装入所缺的页，这取决于这个页是否被映射到磁盘文件。该函数通过检查 vma 虚拟区描述符的 nopage 域来确定这一点，如果页与文件建立起了映射关系，则 nopage 域就指向一个把所缺的页从磁盘装入到 RAM 的函数。因此，可能的情况如下所述。

(1) vma->vm_ops->nopage 域不为 NULL。在这种情况下，某个虚拟区映射一个磁盘文件，nopage 域指向从磁盘读入的函数。这种情况涉及到磁盘文件的低层操作。

(2) 或者 vm_ops 域为 NULL，或者 vma->vm_ops->nopage 域为 NULL。在这种情况下，虚拟区没有映射磁盘文件，也就是说，它是一个匿名映射。因此，do_no_page( )调用 do_anonymous_page( )函数获得一个新的页面：

```
    if (!vma->vm_ops || !vma->vm_ops->nopage)
          return do_anonymous_page (tsk, vma, page_table,
write_access) ;
```

do_anonymous_page（ ）函数分别处理写请求和读请求：

```
if (write_access) {
    page = __get_free_page (GFP_USER) ;
    memset ( (void *) (page) , 0, PAGE_SIZE)
   entry = pte_mkwrite (pte_mkdirty (mk_pte (page,
            vma->vm_page_prot) ) ) ;
    vma->vm_mm->rss++;
    tsk->min_flt++;
    set_pte (pte, entry) ;
     return 1;
}
```

当处理写访问时，该函数调用__get_free_page( )分配一个新的页面，并利用 memset 宏把新页面填为 0。然后该函数增加 tsk 的 min_flt 域以跟踪由进程引起的次级缺页(这些缺页只需要一个新页面)的数目，再增加进程的内存区结构 vma->vm_mm 的 rss 域以跟踪分配给进程的页面数目。然后页表相应的表项被设为页面的物理地址，并把这个页面标记为可写和脏两个标志。

相反，当处理读访问时，页的内容是无关紧要的，因为进程正在对它进行第一次寻址。给进程一个填充为 0 的页要比给它一个由其他进程填充了信息的旧页更为安全。Linux 在请求调页方面做得更深入一些。没有必要立即给进程分配一个填充为零的新页面，由于我们也可以给它一个现有的称为零页的页，这样可以进一步推迟页面的分配。零页在内核初始化期

间被静态分配，并存放在 empty_zero_page 变量中（一个有 1024 个长整数的数组，并用 0 填充）；它存放在第六个页面中（从物理地址 0x00005000 开始），并且可以通过 ZERO_PAGE 宏来引用。

因此页表表项被设为零页的物理地址：

```
entry = pte_wrprotect (mk_pte (ZERO_PAGE, vma->vm_page_prot) ) ;
set_pte (pte, entry) ;
return 1;
```

由于这个页被标记为不可写，如果进程试图写这个页，则写时复制机制被激活。当且仅当在这个时候，进程才获得一个属于自己的页并对它进行写。这种机制在下一部分进行描述。

6.5.5 写时复制

写时复制技术最初产生于 UNIX 系统，用于实现一种傻瓜式的进程创建：当发出 fork( ) 系统调用时，内核原样复制父进程的整个地址空间并把复制的那一份分配给子进程。这种行为是非常耗时的，因为它需要：

- 为子进程的页表分配页面；
- 为子进程的页分配页面；
- 初始化子进程的页表；
- 把父进程的页复制到子进程相应的页中。

创建一个地址空间的这种方法涉及许多内存访问，消耗许多 CPU 周期，并且完全破坏了高速缓存中的内容。在大多数情况下，这样做常常是毫无意义的，因为许多子进程通过装入一个新的程序开始它们的执行，这样就完全丢弃了所继承的地址空间。

现在的 UNIX 内核（包括 Linux），采用一种更为有效的方法称之为写时复制（或 COW）。这种思想相当简单：父进程和子进程共享页面而不是复制页面。然而，只要页面被共享，它们就不能被修改。无论父进程和子进程何时试图写一个共享的页面，就产生一个错误，这时内核就把这个页复制到一个新的页面中并标记为可写。原来的页面仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页面的唯一属主；如果是，它把这个页面标记为对这个进程是可写的。

Page 结构的 count 域用于跟踪共享相应页面的进程数目。只要进程释放一个页面或者在它上面执行写时复制，它的 count 域就递减；只有当 count 变为 NULL 时，这个页面才被释放。

现在我们讲述 Linux 怎样实现写时复制（COW）。当 handle_pte_fault( )确定“缺页”错误是由请求写一个页面所引起的时（这个页面存在于内存中且是写保护的），它执行以下语句：

```
if (pte_present (pte) ) {
    entry = pte_mkyoung (entry) ;
    set_pte (pte, entry) ;
    flush_tlb_page (vma, address) ;
    if (write_access) {
        if (!pte_write (entry) )
            return do_wp_page (tsk, vma, address, pte) ;
        entry = pte_mkdirty (entry) ;
        set_pte (pte, entry) ;
```

```
        flush_tlb_page (vma, address) ;
        }
    return 1;
}
```

首先，调用 pte_mkyoung( ) 和 set_pte( )函数来设置引起错误的页所对应页表项的访问位。这个设置使页“年轻”并减少它被交换到磁盘上的机会。如果错误由违背写保护而引起的，handle_pte_fault( ) 返回由 do_wp_page( )函数产生的值；否则，则已检测到某一错误情况（例如，用户态地址空间中的页，其 User/Supervisor 标志为 0)，且函数返回 1。

do_wp_page( )函数首先把 page_table 参数所引用的页表表项装入局部变量 pte，然后再获得一个新页面：

```
pte = *page_table;
new_page = __get_free_page (GFP_USER) ;
```

由于页面的分配可能阻塞进程，因此，一旦获得页面，这个函数就在页表表项上执行下面的一致性检查：

- 当进程等待一个空闲的页面时，这个页是否已经被交换出去（pte 和 *page_table 的值不相同)；
- 这个页是否已不在物理内存中（页表表项中页的 Present 标志为 0)；
- 页现在是否可写（页项中页的 Read/Write 标志为 1)。

如果以上情况中的任意一个发生，do_wp_page( )释放以前所获得的页面，并返回 1。

现在，函数更新次级缺页的数目，并把引起错误的页的页描述符指针保存到 page_map 局部变量中。

```
tsk->min_flt++;
page_map = mem_map + MAP_NR (old_page) ;
```

接下来，函数必须确定是否必须真的把这个页复制一份。如果仅有一个进程使用这个页，就无须应用写时复制技术，而且进程应该能够自由地写这个页。因此，这个页面被标记为可写，这样当试图写入的时候就不会再次引起“缺页”错误，以前分配的新的页面也被释放，函数结束并返回 1。这种检查是通过读取 page 结构的 count 域而进行的：

```
if (page_map->count == 1) {
    set_pte (page_table, pte_mkdirty (pte_mkwrite (pte) ) ) ;
    flush_tlb_page (vma, address) ;
    if (new_page)
        free_page (new_page) ;
    return 1;
}
```

相反，如果这个页面由两个或多个进程所共享，函数把旧页面(old_page)的内容复制到新分配的页面(new_page)中：

```
if (old_page == ZERO_PAGE)
    memset ( (void *) new_page, 0, PAGE_SIZE) ;
else
    memcpy ( (void *) new_page, (void *) old_page, PAGE_SIZE) ;
set_pte (page_table, pte_mkwrite (pte_mkdirty (
        mk_pte (new_page, vma->vm_page_prot) ) ) ) ;
flush_tlb_page (vma, address) ;
```

```
__free_page（page_map）;
return 1;
```

如果旧页面是零页面，就使用 memset 宏把新的页面填充为 0。否则，使用 memcpy 宏复制页面的内容。不要求一定要对零页作特殊的处理，但是特殊处理确实能够提高系统的性能，因为它使用很少的地址而保护了微处理器的硬件高速缓存。

然后，用新页面的物理地址更新页表的表项，并把新页面标记为可写和脏。最后，函数调用__free_pages( )减小对旧页面的引用计数。

6.5.6 对本节的几点说明

（1）通过 fork（）建立进程，开始时只有一个页目录和一页左右的可执行页，于是缺页异常会频繁发生。

（2）虚拟地址映射到物理地址，只有在请页时才完成，这时要建立页表和更新页表（页表是动态建立的）。页表不可被换出，不记年龄，它们被内核中保留，只有在 exit 时清除。

（3）在处理页故障的过程中，因为要涉及到磁盘访问等耗时操作，因此操作系统会选择另外一个进程进入执行状态，即进行新一轮调度。

6.6 交换机制

当物理内存出现不足时，Linux 内存管理子系统需要释放部分物理内存页面。这一任务由内核的交换守护进程 kswapd 完成，该内核守护进程实际是一个内核线程，它在内核初始化时启动，并周期地运行。它的任务就是保证系统中具有足够的空闲页面，从而使内存管理子系统能够有效运行。

6.6.1 交换的基本原理

如前所述，每个进程的可以使用的虚存空间很大（3GB），但实际使用的空间并不大，一般不会超过几 MB，大多数情况下只有几十 KB 或几百 KB。可是，当系统的进程数达到几百甚至上千个时，对存储空间的总需求就很大，在这种情况下，一般的物理内存量就很难满足要求。因此，在计算机技术的发展史上很早就有了把内存的内容与一个专用的磁盘空间交换的技术，在 Linux 中，我们把用作交换的磁盘空间叫做交换文件或交换区。

交换技术已经使用了很多年。第 1 个 UNIX 系统内核就监控空闲内存的数量。当空闲内存数量小于一个固定的极限值时，就执行换出操作。换出操作包括把进程的整个地址空间拷贝到磁盘上。反之，当调度算法选择出一个进程运行时，整个进程又被从磁盘中交换进来。

现代的 UNIX（包括 Linux）内核已经摒弃了这种方法，主要是因为当进行换入换出时，上下文切换的代价相当高。在 Linux 中，交换的单位是页面而不是进程。尽管交换的单位是页面，但交换还是要付出一定的代价，尤其是时间的代价。实际上，在操作系统中，时间和空间是一对矛盾，常常需要在二者之间作出平衡，有时需要以空间换时间，有时需要以时间

换空间，页面交换就是典型的以时间换空间。这里要说明的是，页面交换是不得已而为之，例如在时间要求比较紧急的实时系统中，是不宜采用页面交换机制的，因为它使程序的执行在时间上有了较大的不确定性。因此，Linux 给用户提供了一种选择，可以通过命令或系统调用开启或关闭交换机制。

在页面交换中，页面置换算法是影响交换性能的关键性指标，其复杂性主要与换出有关。具体说来，必须考虑 4 个主要问题：

- 哪种页面要换出；

如何在交换区中存放页面；

如何选择被交换出的页面；

何时执行页面换出操作。

请注意，我们在这里所提到的页或页面指的是其中存放的数据，因此，所谓页面的换入换出实际上是指页面中数据的换入换出。

1．哪种页面被换出

实际上，交换的最终目的是页面的回收。并非内存中的所有页面都是可以交换出去的。事实上，只有与用户空间建立了映射关系的物理页面才会被换出去，而内核空间中内核所占的页面则常驻内存。我们下面对用户空间中的页面和内核空间中的页面给出进一步的分类讨论。

可以把用户空间中的页面按其内容和性质分为以下几种：

(1) 进程映像所占的页面，包括进程的代码段、数据段、堆栈段以及动态分配的“存储堆”(参见图 6.18)；

(2) 通过系统调用 mmap()把文件的内容映射到用户空间；

(3) 进程间共享内存区。

对于第 1 种情况，进程的代码段数据段所占的内存页面可以被换入换出，但堆栈所占的页面一般不被换出，因为这样可以简化内核的设计。

对于第 2 种情况，这些页面所使用的交换区就是被映射的文件本身。

对于第 3 种情况，其页面的换入换出比较复杂。

与此相对照，映射到内核空间中的页面都不会被换出。具体来说，内核代码和内核中的全局量所占的内存页面既不需要分配(启动时被装入)，也不会被释放，这部分空间是静态的。(相比之下，进程的代码段和全局量都在用户空间，所占的内存页面都是动态的，使用前要经过分配，最后都会被释放，中途可能被换出而回收后另行分配)

除此之外，内核在执行过程中使用的页面要经过动态分配，但永驻内存，此类页面根据其内容和性质可以分为两类。

(1) 内核调用 kmalloc()或 vmalloc()为内核中临时使用的数据结构而分配的页于是立即释放。但是，由于一个页面中存放有多个同种类型的数据结构，所以要到整个页面都空闲时才把该页面释放。

(2) 内核中通过调用 alloc_pages()，为某些临时使用和管理目的而分配的页面，例如，每个进程的内核栈所占的两个页面、从内核空间复制参数时所使用的页面等。这些页面也是一旦使用完毕便无保存价值，所以立即释放。

在内核中还有一种页面，虽然使用完毕，但其内容仍有保存价值，因此，并不立即释放。这类页面“释放”之后进入一个 LRU 队列，经过一段时间的缓冲让其“老化”。如果在此期间又要用到其内容了，就又将其投入使用，否则便继续让其老化，直到条件不再允许时才加以回收。这种用途的内核页面大致有以下这些：

- 文件系统中用来缓冲存储一些文件目录结构 dentry 的空间；
- 文件系统中用来缓冲存储一些索引节点 inode 的空间；
- 用于文件系统读／写操作的缓冲区。

2．如何在交换区中存放页面

我们知道物理内存被划分为若干页面，每个页面的大小为 4KB。实际上，交换区也被划分为块，每个块的大小正好等于一页，我们把交换区中的一块叫做一个页插槽（Page Slot），意思是说，把一个物理页面插入到一个插槽中。当进行换出时，内核尽可能把换出的页放在相邻的插槽中，从而减少在访问交换区时磁盘的寻道时间。这是高效的页面置换算法的物质基础。

如果系统使用了多个交换区，事情就变得更加复杂了。快速交换区（也就是存放在快速磁盘中的交换区）可以获得比较高的优先级。当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索。如果优先级最高的交换区不止一个，为了避免超负荷地使用其中一个，应该循环选择相同优先级的交换区。如果在优先级最高的交换区中没有找到空闲插槽，就在优先级次高的交换区中继续进行搜索，依此类推。

3．如何选择被交换出的页面

页面交换是非常复杂的，其主要内容之一就是如何选择要换出的页面，我们以循序渐进的方式来讨论页面交换策略的选择。

策略一，需要时才交换。每当缺页异常发生时，就给它分配一个物理页面。如果发现没有空闲的页面可供分配，就设法将一个或多个内存页面换出到磁盘上，从而腾出一些内存页面来。这种交换策略确实简单，但有一个明显的缺点，这是一种被动的交换策略，需要时才交换，系统势必要付出相当多的时间进行换入换出。

策略二，系统空闲时交换。与策略一相比较，这是一种积极的交换策略，也就是，在系统空闲时，预先换出一些页面而腾出一些内存页面，从而在内存中维持一定的空闲页面供应量，使得在缺页中断发生时总有空闲页面可供使用。至于换出页面的选择，一般都采用 LRU（最近最少使用）算法。但是这种策略实施起来也有困难，因为并没有哪种方法能准确地预测对页面的访问，所以，完全可能发生这样的情况，即一个好久没有受到访问的页面刚刚被换出去，却又要访问它了，于是又把它换进来。在最坏的情况下，有可能整个系统的处理能力都被这样的换入／换出所影响，而根本不能进行有效的计算和操作。这种现象被称为页面的“抖动”。

策略三，换出但并不立即释放。当系统挑选出若干页面进行换出时，将相应的页面写入磁盘交换区中，并修改相应页表中页表项的内容（把 present 标志位置为 0），但是并不立即释放，而是将其 page 结构留在一个缓冲（Cache）队列中，使其从活跃（Active）状态转为

不活跃（Inactive）状态。至于这些页面的最后释放，要推迟到必要时才进行。这样，如果一个页面在释放后又立即受到访问，就可以从物理页面的缓冲队列中找到相应的页面，再次为之建立映射。由于此页面尚未释放，还保留着原来的内容，就不需要磁盘读入了。经过一段时间以后，一个不活跃的内存页面一直没有受到访问，那这个页面就需要真正被释放了。

策略四，把页面换出推迟到不能再推迟为止。实际上，策略三还有值得改进的地方。首先在换出页面时不一定要把它的内容写入磁盘。如果一个页面自从最近一次换入后并没有被写过（如代码），那么这个页面是“干净的”，就没有必要把它写入磁盘。其次，即使“脏”页面，也没有必要立即写出去，可以采用策略三。至于“干净”页面，可以一直缓冲到必要时才加以回收，因为回收一个“干净”页面花费的代价很小。

下面对物理页面的换入/换出给出一个概要描述，这里涉及到前面介绍的 page 结构和 free_area 结构。

（1）释放页面。如果一个页面变为空闲可用，就把该页面的 page 结构链入某个页面管理区(Zone)的空闲队列 free_area，同时页面的使用计数 count 减 1。

（2）分配页面。调用__alloc_pages()或__get_free_page()从某个空闲队列分配内存页面，并将其页面的使用计数 count 置为 1。

（3）活跃状态。已分配的页面处于活跃状态，该页面的数据结构 page 通过其队列头结构 lru 链入活跃页面队列 active_list，并且在进程地址空间中至少有一个页与该页面之间建立了映射关系。

（4）不活跃“脏”状态。处于该状态的页面其 page 结构通过其队列头结构 lru 链入不活跃“脏”页面队列 inactive_dirty_list，并且原则是任何进程的页面表项不再指向该页面，也就是说，断开页面的映射，同时把页面的使用计数 count 减 1。

（5）将不活跃“脏”页面的内容写入交换区，并将该页面的 page 结构从不活跃“脏”页面队列 inactive_dirty_list 转移到不活跃“干净”页面队列，准备被回收。

（6）不活跃“干净”状态。页面 page 结构通过其队列头结构 lru 链入某个不活跃“干净”页面队列，每个页面管理区都有个不活跃“干净”页面队列 inactive_clean_list。

（7）如果在转入不活跃状态以后的一段时间内，页面又受到访问，则又转入活跃状态并恢复映射。

（8）当需要时，就从“干净”页面队列中回收页面，也就是说或者把页面链入到空闲队列，或者直接进行分配。

以上是页面换入／换出及回收的基本思想，实际的实现代码还要更复杂一些。

4．何时执行页面换出操作

为了避免在 CPU 忙碌的时候，也就是在缺页异常发生时，临时搜索可供换出的内存页面并加以换出，Linux 内核定期地检查系统内的空闲页面数是否小于预定义的极限，一旦发现空闲页面数太少，就预先将若干页面换出，以减轻缺页异常发生时系统所承受的负担。当然，由于无法确切地预测页面的使用，即使这样做了也还可能出现缺页异常发生时内存依然没有足够的空闲页面。但是，预换出毕竟能减少空闲页面不够用的概率。并且通过选择适当的参数（如每隔多久换出一次，每次换出多少页），可以使临时寻找要换出页面的情况很少发生。为此，Linux 内核设置了一个专伺定期将页面换出的守护进程 kswapd。

6.6.2 页面交换守护进程 kswapd

从原理上说，kswapd 相当于一个进程，它有自己的进程控制块 task_struct 结构。与其他进程一样受内核的调度。而正因为内核将它按进程来调度，就可以让它在系统相对空闲的时候来运行。不过，与普通进程相比，kswapd 有其特殊性。首先，它没有自己独立的地址空间，所以在近代操作系统理论中把它称为“线程”以与进程相区别。那么，kswapd 的地址空间是什么？实际上，内核空间就是它的地址空间。在这一点上，它与中断服务例程相似。其次，它的代码是静态地链接在内核中的，因此，可以直接调用内核中的各种子程序和函数。

kswapd 的源代码基本上都在 mm/vmscan.c 中，图 6.19 给出了 kswapd 中与交换有关的主要函数调用关系。

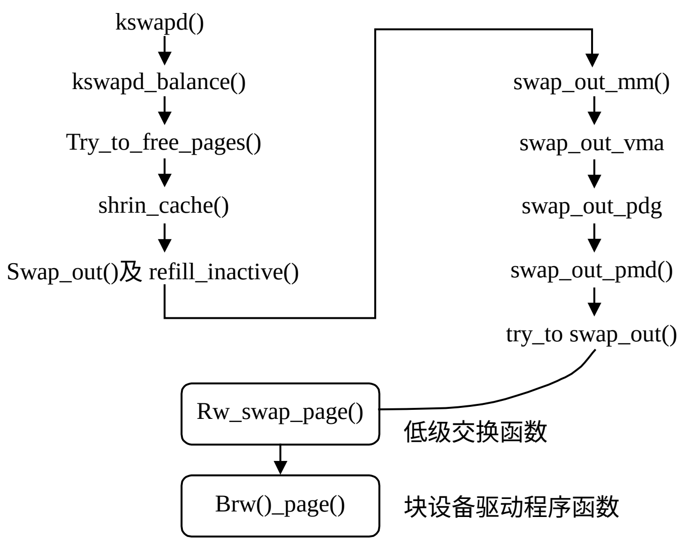


图 6.19 kswapd 的实现代码中与交换相关的主要函数的调用关系

从上面的调用关系可以看出，kswapd 的实现相当复杂，这不仅仅涉及复杂的页面交换技术，还涉及与磁盘相关的具体文件操作，因此，为了理清思路，搞清主要内容，我们对一些主要函数给予描述。

1. kswapd（）

在 Linux 2.4.10 以后的版本中对 kswapd（）的实现代码进行了模块化组织，可读性大大加强，代码如下：

```
int kswapd（void *unused)
{
        struct task_struct *tsk = current;
        DECLARE_WAITQUEUE（wait, tsk）;

        daemonize（）;          /*内核线程的初始化*/
        strcpy（tsk->comm, "kswapd"）;
        sigfillset（&tsk->blocked）;  /*把进程 PCB 中的阻塞标志位全部置为 1*/
```

```
        /*
         * Tell the memory management that we're a "memory allocator",
        * and that if we need more memory we should get access to it
         * regardless  (see "__alloc_pages () ") . "kswapd" should
         * never get caught in the normal page freeing logic.
        *
        *  (Kswapd normally doesn't need memory anyway, but sometimes
         * you need a small amount of memory in order to be able to
         * page out something else, and this flag essentially protects
         * us from recursively trying to free more memory as we're
         * trying to free the first piece of memory in the first place) .
        */
        tsk->flags |= PF_MEMALLOC; ／*这个标志表示给 kswapd 要留一定的内存*/

        /*
         * Kswapd main loop.
         */
        for  (;;)  {
               __set_current_state (TASK_INTERRUPTIBLE) ;
               add_wait_queue (&kswapd_wait, &wait) ; ／*把 kswapd 加入等待队列*／

               mb () ;   ／*增加一条汇编指令*/
                if  (kswapd_can_sleep () )     ／*检查调度标志是否置位*/
                        schedule () ;     ／*调用调度程序*/

            _set_current_state (TASK_RUNNING) ; ／*让 kswapd 处于就绪状态*/
            remove_wait_queue (&kswapd_wait, &wait) ; ／*把 kswapd 从等待队列删除*/

                /*
                 * If we actually get into a low-memory situation,
                 * the processes needing more memory will wake us
                 * up on a more timely basis.
                */
                kswapd_balance () ;   ／* kswapd 的核心函数，请看后面内容*/
               run_task_queue (&tq_disk) ;  ／*运行 tq_disk 队列中的例程*/
        }
}
```

kswapd 是内存管理中唯一的一个线程，其创建如下：

```
static int __init kswapd_init (void)
{
        printk ("Starting kswapd\n") ;
        swap_setup () ;
        kernel_thread (kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL) ;
       return 0;
}
```

然后，在内核启动时由模块的初始化例程调用 kswapd_init：

```
module_init(kswapd_init)
```

从上面的介绍可以看出，kswapd 成为内核的一个线程，其主循环是一个无限循环。循环一开始，把它加入等待队列，但如果调度标志为 1，就执行调度程序，紧接着就又把它从等

待队列删除，将其状态变为就绪。只要调度程序再次执行，它就会得到执行，如此周而复始进行下去。

2．kswapd_balance()函数

从该函数的名字可以看出，这是一个要求得平衡的函数，那么，求得什么样的平衡呢？在本章的初始化一节中，我们介绍了物理内存的 3 个层次，即存储节点、管理区和页面。所谓平衡就是对页面的释放要均衡地在各个存储节点、管理区中进行，代码如下：

```
static void kswapd_balance (void)
{
         int need_more_balance;
         pg_data_t * pgdat;

         do {
                 need_more_balance = 0;
                 pgdat = pgdat_list;
                do
                         need_more_balance |= kswapd_balance_pgdat (pgdat) ;
                while  ( (pgdat = pgdat->node_next) ) ;
        } while  (need_more_balance) ;
}
```

这个函数比较简单，主要是对每个存储节点进行扫描。然后又调用 kswapd_balance_pgdat（）对每个管理区进行扫描：

```
static int kswapd_balance_pgdat (pg_data_t * pgdat)
{
         int need_more_balance = 0, i;
        zone_t * zone;

         for  (i = pgdat->nr_zones-1; i >= 0; i--)  {
                zone = pgdat->node_zones + i;
                 if  (unlikely (current->need_resched) )
                         schedule () ;
                 if  (!zone->need_balance)
                         continue;
                 if  (!try_to_free_pages (zone, GFP_KSWAPD, 0) )  {
                         zone->need_balance = 0;
                        __set_current_state (TASK_INTERRUPTIBLE) ;
                         schedule_timeout (HZ) ;
                        continue;
                 }
                if  (check_classzone_need_balance (zone) )
                         need_more_balance = 1;
                 else
                         zone->need_balance = 0;
         }
```

其中，最主要的函数是 try_to_free_pages（），能否调用这个函数取决于平衡标志 need_balance 是否为 1，也就是说看某个管理区的空闲页面数是否小于最高警戒线，这是由 check_classzone_need_balance()函数决定的。当某个管理区的空闲页面数小于其最高警戒

线时就调用 try_to_free_pages（）。

3．try_to_free_pages（）

该函数代码如下：

```
   int try_to_free_pages (zone_t *classzone, unsigned int gfp_mask, unsigned int order)
{
        int priority = DEF_PRIORITY;
        int nr_pages = SWAP_CLUSTER_MAX;

        gfp_mask = pf_gfp_mask (gfp_mask) ;
        do {
              nr_pages = shrink_caches (classzone, priority, gfp_mask, nr_pages) ;
               if  (nr_pages <= 0)
                     return 1;
        } while  (--priority) ;

        /*
          * Hmm.. Cache shrink failed - time to kill something?
          * Mhwahahhaha! This is the part I really like. Giggle.
          */
        out_of_memory () ;
        return 0;
}
```

其中的优先级表示对队列进行扫描的长度，缺省的优先级 DEF_PRIORITY 为 6（最低优先级）。假定队列长度为 L，优先级 6 就表示要扫描的队列长度为 L／2^6，所以这个循环至少循环 6 次。nr_pages 为要换出的页面数，其最大值 SWAP_CLUSTER_MAX 为 32。其中主要调用的函数为 shrink_caches（）：

```
   static int shrink_caches (zone_t * classzone, int priority, unsigned int gfp_mask, int
nr_pages)
{
        int chunk_size = nr_pages;
        unsigned long ratio;

        nr_pages -= kmem_cache_reap (gfp_mask) ;
        if  (nr_pages <= 0)
               return 0;

        nr_pages = chunk_size;
        /* try to keep the active list 2/3 of the size of the cache */
        ratio =  (unsigned long)  nr_pages * nr_active_pages /  ( (nr_inactive_pages + 1)
* 2) ;
        refill_inactive (ratio) ;

        nr_pages = shrink_cache (nr_pages, classzone, gfp_mask, priority) ;
        if  (nr_pages <= 0)
               return 0;

        shrink_dcache_memory (priority, gfp_mask) ;
        shrink_icache_memory (priority, gfp_mask) ;
```

```
1 #ifdef CONFIG_QUOTA
        shrink_dqcache_memory (DEF_PRIORITY, gfp_mask) ;
#endif

        return nr_pages;
}
```

其中 kmem_cache_reap（）函数“收割(reap)”由 slab 机制管理的空闲页面。如果从 slap 回收的页面数已经达到要换出的页面数 nr_pages，就不用从其他地方进行换出。refill_inactive（）函数把活跃队列中的页面移到非活跃队列。shrink_cache（）函数把一个“洗净”且未加锁的页面移到非活跃队列，以便该页能被尽快释放。

此外，除了从各个进程的用户空间所映射的物理页面中回收页面外，还调用 shrink_dcache_memory（）、shrink_icache_memory（）及 shrink_dqcache_memory（）回收内核数据结构所占用的空间。在文件系统一章将会看到，在打开文件的过程中，要分配和使用代表着目录项的 dentry 数据结构，还有代表着文件索引节点 inode 的数据结构。这些数据结构在文件关闭后并不立即释放，而是放在 LRU 队列中作为后备，以防在将来的文件操作中又用到。这样经过一段时间后，就有可能积累起大量的 dentry 数据结构和 inode 数据结构，从而占用数量可观的物理页面。这时，就要通过这些函数适当加以回收。

4．页面置换

到底哪些页面会被作为后选页以备换出，这是由 swap_out（）和 shrink_cache（）一起完成的。这个过程比较复杂，这里我们抛开源代码，以理清思路为目标。

shrink_cache（）要做很多换出的准备工作。它关注两个队列：“活跃的” LRU 队列 和“非活跃的” FIFO 队列，每个队列都是 struct page 形成的链表。该函数的代码比较长，我们把它所做的工作概述如下：

- 把引用过的页面从活跃队列的队尾移到该队列的队头（实现 LRU 策略）；
- 把未引用过的页面从活跃队列的队尾移到非活跃队列的队头（为准备换出而排队）；
- 把脏页面安排在非活跃队列的队尾准备写到磁盘；
- 从非活跃队列的队尾恢复干净页面(写出的页面就成为干净的)。

6.6.3 交换空间的数据结构

Linux 支持多个交换文件或设备，它们将被 swapon 和 swapoff 系统调用来打开或关闭。每个交换文件或设备都可用 swap_info_struct 结构来描述：

```
struct swap_info_struct {
        unsigned int flags;
        kdev_t swap_device;
        spinlock_t sdev_lock;
        struct dentry * swap_file;
        struct vfsmount *swap_vfsmnt;
        unsigned short * swap_map;
        unsigned int lowest_bit;
        unsigned int highest_bit;
        unsigned int cluster_next;
```

```
        unsigned int cluster_nr;
        int prio;                    /* swap priority */
        int pages;
        unsigned long max;
        int next;                    /* next entry on swap list */
};

extern int nr_swap_pages;
```

flags 域(SWP_USED 或 SWP_WRITEOK)用作控制访问交换文件。当 swapoff 被调用（为了取消一个文件）时，SWP_WRITEOK 置成 off，使在文件中无法分配空间。如果 swapon 加入一个新的交换文件时，SWP_USED 被置位。这里还有一静态变量(nr_swapfiles)来记录当前活动的交换文件数。

域 lowest_bit，highest_bit 表明在交换文件中空闲范围的边界，这是为了快速寻址。

当用户程序 mkswap 初始化交换文件或设备时，在文件的第一个页插槽的前 10 个字节，有一个包含有位图的标志，在位图里初始化为 0，代表坏的页插槽，1 代表相关页插槽是空闲的。

当用户程序调用 swapon()时，有一页被分配给 swap_map。

swap_map 为在交换文件中每一个页插槽保留了一个字节，0 代表可用页插槽，128 代表不可用页插槽。它被用于记下交换文件中每一页插槽上的 swap 请求。

内存中的一页被换出时，调用 get_swap_page（）会得到一个一个记录换出位置的索引，然后在页表项中回填（1～31 位）此索引。这是为了在发生在缺页异常时进行处理(do_no_page)。索引的高 7 位给定交换文件，后 24 位给定设备中的页插槽号。

另外函数 swap_duplicate()被 copy_page_tables()调用来实现子进程在 fork()时继承被换出的页面，这里要增加域 swap_map 中此页面的 count 值，任何进程访问此页面时，会换入它的独立的拷贝。

swap_free()减少域 swap_map 中的 count 值，如果 count 减到 0 时，则这页面又可再次分配（get_swap_page），在把一个换出页面调入(swap_in)内存时或放弃一个页面时(free_one_table）调用 swap_free()。

相关函数在文件 filemap.c 中。

6.6.4　交换空间的应用

1. 建立交换空间

作为交换空间的交换文件实际就是通常的文件，但文件的扇区必须是连续的，即文件中必须没有“洞”，另外，交换文件必须保存在本地硬盘上。

由于内核要利用交换空间进行快速的内存页面交换，因此，它不进行任何文件扇区的检查，而认为扇区是连续的。由于这一原因，交换文件不能包含洞。可用下面的命令建立无洞的交换文件：

```
$ dd if=/dev/zero of=/extra-swap bs=1024 count=2048
2048+0 records in
```

```
2048+0 records out
```

上面的命令建立了一个名称为 extra-swap，大小为 2048KB 的交换文件。对 i386 系统而言，由于其页面尺寸为 4KB，因此最好建立一个大小为 4K 倍数的交换文件；对 Alpha AXP 系统而言，最好建立大小为 8K 倍数的交换文件。

交换分区和其他分区也没有什么不同，可像建立其他分区一样建立交换分区。但该分区不包含任何文件系统。分区类型对内核来讲并不重要，但最好设置为 Linux Swap 类型（即类型 82）。

建立交换文件或交换分区之后，需要在文件或分区的开头写入签名，写入的签名实际是由内核使用的一些管理信息。写入签名的命令为 mkswap，如下所示：

```
$ mkswap /extra-swp 2048
Setting up swapspace, size = 2088960 bytes
$
```

这时，新建立的交换空间尚未开始使用。使用 mkswap 命令时必须小心，因为该命令不会检查文件或分区内容，因此极有可能覆盖有用的信息，或破坏分区上的有效文件系统信息。

Linux 内存管理子系统将每个交换空间的大小限制在 127MB （实际为 (4096.10)* 8*4096 = 133890048 Byte = 127.6875MB）。可以在系统中同时使用 16 个交换空间，从而使交换空间总量达到 2GB。

2．使用交换空间

利用 swapon 命令可将经过初始化的交换空间投入使用。如下所示：

```
$ swapon /extra-swap
$
```

如果在 /etc/fstab 文件中列出交换空间，则可自动将交换空间投入使用：

```
/dev/hda5 none swap sw 0 0
/extra-swap none swap sw 0 0
```

实际上，启动脚本会运行 swapon –a 命令，从而将所有出现在 /etc/fstab 文件中的交换空间投入使用。

利用 free 命令，可查看交换空间的使用。如下所示：

```
$ free
total used free shared buffers
Mem: 15152 14896 256 12404 2528
-/+ buffers: 12368 2784
Swap: 32452 6684 25768
$
```

该命令输出的第一行（Mem：） 显示了系统中物理内存的使用情况。total 列显示的是系统中的物理内存总量；used 列显示正在使用的内存数量；free 列显示空闲的内存量；shared 列显示由多个进程共享的内存量，该内存量越多越好；buffers 显示了当前的缓冲区高速缓存的大小。

输出的最后一行 （Swap：） 显示了有关交换空间的类似信息。如果该行的内容均为 0，表明当前没有活动的交换空间。

利用 top 命令或查看 /proc 文件系统中的 /proc/meminfo 文件可获得相同的信息。

利用 swapoff 命令可移去使用中的交换空间。但该命令应只用于临时交换空间，否则

有可能造成系统崩溃。

swapoff –a 命令按照 /etc/fstab 文件中的内容移去所有的交换空间，但任何手工投入使用的交换空间保留不变。

3．分配交换空间

大多数人认为，交换空间的总量应该是系统物理内存量的两倍，实际上这一规则是不正确的，正确的交换空间大小应按如下规则确定。

（1）估计需要的内存总量。运行想同时运行的所有程序，并利用 free 或 ps 程序估计所需的内存总量，只需大概估计。

（2）增加一些安全性余量。

（3）减去已有的物理内存数量，然后将所得数据取整为 MB，这就是应当的交换空间大小。

（4）如果得到的交换空间大小远远大于物理内存量，则说明需要增加物理内存数量，否则系统性能会因为过分的页面交换而下降。

（5）当计算的结果说明不需要任何交换空间时，也有必要使用交换空间。Linux 从性能的角度出发，会在磁盘空闲时将某些页面交换到交换空间中，以便减少必要时的交换时间。另外，如果在不同的磁盘上建立多个交换空间，有可能提高页面交换的速度，这是因为某些硬盘驱动器可同时在不同的磁盘上进行读写操作。

6.7　缓存和刷新机制

6.7.1　Linux 使用的缓存

不管在硬件设计还是软件设计中，高速缓存是获得高性能的常用手段。Linux 使用了多种和内存管理相关的高速缓存。

1．缓冲区高速缓存

缓冲区高速缓存中包含了由块设备使用的数据缓冲区。这些缓冲区中包含了从设备中读取的数据块或写入设备的数据块。缓冲区高速缓存由设备标识号和块标号索引，因此可以快速找出数据块。如果数据能够在缓冲区高速缓存中找到，则系统就没有必要在物理块设备上进行实际的读操作。

内核为每个缓冲区维护很多信息以有助于缓和写操作，这些信息包括一个“脏（dirty）”位，表示内存中的缓冲区已被修改，必须写到磁盘；还包括一个时间标志，表示缓冲区被刷新到磁盘之前已经在内存中停留了多长时间。因为缓冲区的有关信息被保存在缓冲区首部，所以，这些数据结构连同用户数据本身的缓冲区都需要维护。

缓冲区高速缓存的大小可以变化。当需要新缓冲区而现在又没有可用的缓冲区时，就按需分配页面。当空闲内存变得不足时，例如上一节看到的情况，就释放缓冲区并反复使用相

应的页面。

2．页面高速缓存

页面高速缓存是页面 I/0 操作访问数据所使用的磁盘高速缓存。我们在文件系统会看到，read( )、write( )和 mmap( )系统调用对常规文件的访问都是通过页面高速缓存来完成的。因为页面 I/0 操作要传输整页数据，因此高速缓存中所保留的信息单元是一个整页面。一个页面包含的数据未必是物理上相邻的磁盘块，因此就不能使用设备号和块号来标识页面。相反，页面高速缓存中一个页面的标识是通过文件的索引节点和文件中的偏移量达到的。

与页面高速缓存有关的操作主要有 3 种：①当访问的文件部分不在高速缓存中时增加一页面；②当高速缓存变得太大时删除一页面；③查找一个给定文件偏移量所在的页面。

3．交换高速缓存

只有修改后的（脏）页面才保存在交换文件中。修改后的页面写入交换文件后，如果该页面再次被交换但未被修改时，就没有必要写入交换文件，相反，只需丢弃该页面。交换高速缓存实际包含了一个页面表项链表，系统的每个物理页面对应一个页面表项。对交换出的页面，该页面表项包含保存该页面的交换文件信息，以及该页面在交换文件中的位置信息。如果某个交换页面表项非零，则表明保存在交换文件中的对应物理页面没有被修改。如果这一页面在后续的操作中被修改，则处于交换缓存中的页面表项被清零。 Linux 需要从物理内存中交换出某个页面时，它首先分析交换缓存中的信息，如果缓存中包含该物理页面的一个非零页面表项，则说明该页面交换出内存后还没有被修改过，这时，系统只需丢弃该页面。

这里给出有关交换缓存的部分函数及功能：位于/ linux/mm/swap_state.c 中。

初始化交换缓冲，设定大小，位置的函数：

```
extern unsigned long init_swap_cache (unsigned long, unsigned long) ;
```

显示交换缓冲信息的函数：

```
extern void show_swap_cache_info (void) ;
```

加入交换缓冲的函数：

```
int add_to_swap_cache (unsigned long index, unsigned long entry)
```

参数 index 是进入缓冲区的索引（index 是索引表中的某一项），entry 是‘页面表项’（即此页面在交换文件中的位置记录，这个记录类似页面表项，参见交换机制）

复制被换出的页面：

```
extern void swap_duplicate (unsigned long) ;
```

当使用 copy_page_tables()调用，来实现子进程在 fork()时继承被换出的页面，可参阅交换机制一节。

从缓冲区中移去某页面

```
delete_from_swap_cache (page_nr) ;
```

硬件高速缓存：

常见的硬件缓存是对页面表项的缓存，这一工作实际由处理器完成，其操作和具体的处理器硬件有关（但管理要由软件完成），对这一缓存接下来要做进一步描述。

6.7.2　缓冲区高速缓存

Linux 采用了缓冲区高速缓存机制，而不同于其他操作系统的“写透”方式，也就是说，当把一个数据写入文件时，内核将把数据写入内存缓冲区，而不是直接写入磁盘。

在这里要用到一个数据结构 buffer_head，它是用来描述缓冲区的数据结构，缓冲区的大小一般要比页面尺寸小，所以一页面中可以包含数个缓冲区，同一页面中的缓冲区用链表连接。回忆一下页面结构 page，其中有一个域 buffer_head buffer 就是用来指向缓冲区的，这个结构的详细内容请参见虚拟文件系统。

由于使用了缓冲技术，因此有可能出现这种情况：写磁盘的命令已经返回，但实际的写入磁盘的操作还未执行。

基于上述原因，应当使用正常的关机命令关机，而不应直接关掉计算机的电源。用户也可以使用 sync 命令刷新缓冲区高速缓存，从而把缓冲区中的数据强制写到磁盘中。在 Linux 系统中，除了传统的 update 守护进程之外，还有一个额外的守护进程 dbflush，这一进程可频繁运行不完整的 sync 从而可避免有时由于 sync 命令的超负荷磁盘操作而造成的磁盘冻结，一般情况下，它们在系统引导时自动执行，且每隔 30s 执行一次任务。

sync 命令使用基本的系统调用 sync()来实现。

dbflush 在 Linux 系统中由 update 启动。如果由于某种原因该进程僵死了，则内核会发送警告信息，这时需要手工启动该进程（/sbin/update）。

1．页面缓存的详细描述

经内存映射的文件每次只读取一页面内容，读取后的页面保存在页面缓存中，利用页面缓存，可提高文件的访问速度。如图 6.20 所示，页面缓存由 page_hash_table 组成，它是一个 mem_map_t(即 struct page 数据结构)的指针向量。页面缓存的结构是 Linux 内核中典型的哈希表结构。众所周知，对计算机内存的线性数组的访问是最快速的访问方法，因为线性数组中的每一个元素的位置都可以利用索引值直接计算得到，而这种计算是简单的线性计算。但是，如果要处理大量数据，有时由于受到存储空间的限制，采用线性结构是不切合实际的。但如果采用链表等非线性结构，则元素的检索性能又会大打折扣。哈希表则是一种折衷的方法，它综合了线性结构和非线性结构的优点，可以在大量数据中进行快速的查找。哈希表的结构有多种，在 Linux 内核中，常见的哈希结构和图 6.20 的结构类似。要在这种哈希表中访问某个数据，首先要利用哈希函数以目标元素的某个特征值作为函数自变量生成哈希值作为索引，然后利用该索引访问哈希表的线性指针向量。哈希线性表中的指针代表一个链表，该链表所包含的所有节点均具有相同的哈希值，在该链表中查找可访问到指定的数据。哈希函数的选择非常重要，不恰当的哈希函数可能导致大量数据映射到同一哈希值，这种情况下，元素的查找将相当耗时。但是，如果选择恰当的哈希函数，则可以在性能和空间上得到均衡效果。

在 Linux 页面缓存中，访问 page_hash_table 的索引由文件的 VFS（虚拟文件系统）索引节点 inode 和内存页面在文件中的偏移量生成。有关 VFS 索引节点的内容将在虚拟文件中讲到，在这里，应知道每个文件的 VFS 索引节点 inode 是唯一的。

当系统要从内存映射文件中读取某一未加锁的页面时，就首先要用到函数：

find_page (struct inode * inode, unsigned long offset)。

它完成如下工作。

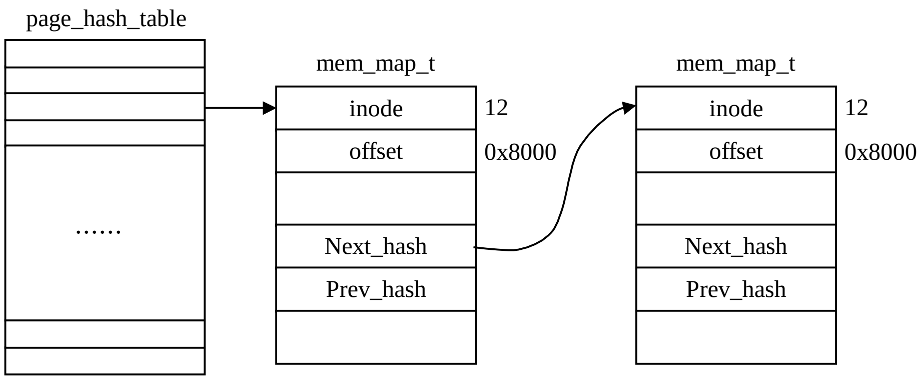


图 6.20 Linux 页面缓存示意图

首先是在“页面缓存”中查找，如果发现该页面保存在缓存中，则可以免除实际的文件读取，而只需从页面缓存中读取，这时，指向 mm_map_t 数据结构的指针被返回到页面故障的处理代码。部分代码如下：

```
for (page = page_hash (inode, offset) ; page ; page = page->next_hash)

/*函数 page_hash () 是从哈希表中找页面*/
{if (page->inode != inode)
     continue;
if (page->offset != offset)
    continue;
    /* 找到了特定页面 */
    atomic_inc (&page->count) ;
    set_bit (PG_referenced, &page->flags) ;/*设访问位*/
    break;   }
    return page;
}
```

如果该页面不在缓存中，则必须从实际的文件系统映像中读取页面，这时 Linux 内核首先分配物理页面然后从磁盘读取页面内容。

如果可能，Linux 还会预先读取文件中下一页面内容到页面缓存中，而不等页面错误发生才去“请页面”，这样做是为了提高装入代码的速度(有关代码在 filemap.c 中，如 generic_file_readahead () 等函数)。这样，如果进程要连续访问页面，则下一页面的内容不必再次从文件中读取了，而只需从页面缓存中读取。

随着映像的读取和执行，页面缓存中的内容可能会增多，这时，Linux 可移走不再需要的页面。当系统中可用的物理内存量变小时，Linux 也会通过缩小页面缓存的大小而释放更多的物理内存页面。

2．有关页面缓存的函数

先看把读入的页面如何存于缓存，这要用到函数 add_to_page_cache()，它完成把指定的“文件页面”记入页面缓存中。

```
static inline void add_to_page_cache (struct page * page,
     struct inode * inode, unsigned long offset)
{    /*设置有关页面域，引用数，页面使用方式，页面在文件中的偏移 */
    page->count++;
     page->flags &= ~ ( (1 << PG_uptodate) | (1 << PG_error) ) ;
     page->offset = offset;
     add_page_to_inode_queue (inode, page) ;/* 把页面加入 inode 节点队列*/
     add_page_to_hash_queue (inode, page) ;/* 把页面加入哈唏表 page_hash_table[]*/}
```

注意：inode 的部分请看虚拟文件章节。

哈希表 page_hash_table[]的定义：

```
extern struct page * page_hash_table[PAGE_HASH_SIZE];
```

下面是有关对哈希表操作的部分代码：

```
static inline void add_page_to_inode_queue (struct inode * inode, struct page * page)
{    struct page **p = &inode->i_pages;/*指向物理页面*/
     inode->i_nrpages++;/*节点中调入内存的页面数目增 1*/
     page->inode = inode; /*指向该页面来自的文件节点结构，相互连成链*/
     page->prev = NULL;
if ( (page->next = *p)  != NULL)
page->next->prev = page;
     *p = page;  }
```

把页面加入哈希表：

```
static inline void add_page_to_hash_queue (struct inode * inode, struct page * page)
{    struct page **p = &page_hash (inode,page->offset) ;
     page_cache_size++;/*哈希表中记录的页面数目加 1*/
     set_bit (PG_referenced, &page->flags) ;/*设置访问位*/
page->age = PAGE_AGE_VALUE;/*设缓存中的页面“年龄”为定值，为淘汰做准
备*/
     page->prev_hash = NULL;
     if ( (page->next_hash = *p)  != NULL)
     page->next_hash->prev_hash = page;    *p = page;
}
```

有关页面的刷新函数：

```
remove_page_from_hash_queue (page) ;   /*从哈希表中去掉页面*/
remove_page_from_inode_queue (page) ;  /*从 inode 节点中去掉页面*/
```

6.7.3 翻译后援存储器(TLB)

页表的实现对虚拟内存系统效率是极为关键的。例如把一个寄存器的内容复制到另一个寄存器中的一条指令，在不使用分页时，只需访问内存一次取指令，而在使用分页时需要额外的内存访问去读取页表。而系统的运行速度一般是被 cpu 从内存中取得指令和数据的速率限制的，如果在每次访问内存时都要访问两次内存会使系统性能降低三分之二。

对这个问题的解决，有人提出了一个解决方案，这个方案基于这样的观察：大部分程序

倾向于对较少的页面进行大量的访问。因此，只有一小部分页表项经常被用到，其他的很少被使用。

采取的解决办法是为计算机装备一个不需要经过页表就能把虚拟地址映射成物理地址的小的硬件设备，这个设备叫做 TLB(翻译后援存储器，Translation Lookside Buffer)，有时也叫做相联存储器(Associative Memory)，如图 6.21 所示 。它通常在 MMU 内部，条目的数量较少，在这个例子中是 6 个，80386 有 32 个。

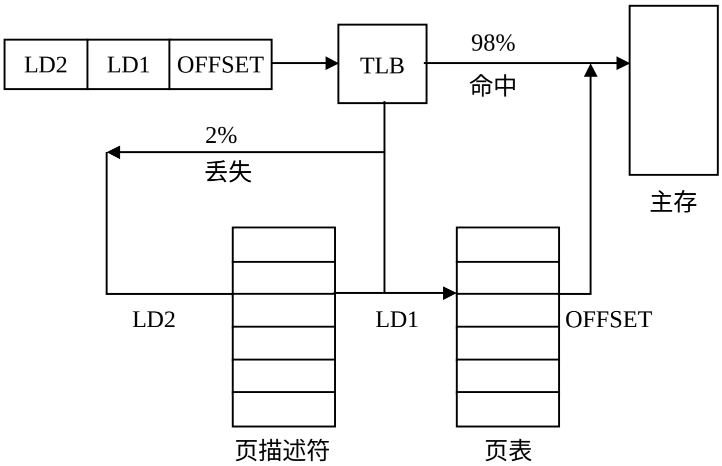


图 6.21　翻译后援存储器

每一个 TLB 寄存器的每个条目包含一个页面的信息：有效位、虚页面号、修改位、保护码和页面所在的物理页面号，它们和页面表中的表项一一对应，如图 6.22 所示。

段号	虚页面号	页面框	保护	年龄	有效位
4	1	7	RW	5	1
8	7	16	RW	1	1
2	0	33	RX	4	1
4	4	72	RX	13	0
5	8	17	RW	2	1
2	7	34	RX	2	1

图 6.22　用于加速分页面操作的 TLB

当一个虚地址被送到 MMU 翻译时，硬件首先把它和 TLB 中的所有条目同时(并行地)进行比较。如果它的虚页面号在 TLB 中，并且访问没有违反保护位，它的页面会直接从 TLB 中取出而不去访问页表；如果虚页面号在 TLB 中，但当前指令试图写一个只读的页面，这时将产

生一个缺页异常，与直接访问页表时相同。

如 MMU 发现在 TLB 中没有命中，它将随即进行一次常规的页表查找，然后从 TLB 中淘汰一个条目并把它替换为刚刚找到的页表项。因此如果这个页面很快再被用到的话，第 2 次访问时它就能在 TLB 中直接找到。在一个 TLB 条目被淘汰时，被修改的位被复制回在内存中的页表项，其他的值则已经在那里了。当 TLB 从页表装入时，所有的域都从内存中取得。

必须明确在分页机制中，TLB 中的数据和页表中的数据的相关性，不是由处理器进行维护，而是必须由操作系统来维护，高速缓存的刷新是通过装入处理器（80386）中的寄存器 CR3 来完成的（见刷新机制 flush_tlb()）。

这里提到的命中率，指一个页面在 TBL 中找到的概率。一般来说 TLB 的尺寸大可增加命中率，但会增加成本和软件的管理。所以一般都采用 8～64 个条目的数量。

假如命中率是 0.85，访问内存时间是 120 纳秒，访 TLB 时间是 15 纳秒。那么访问时间是：0.85×(15+120)+(1-0.85)×(15+120+120)=153 纳秒。

6.7.4　刷新机制

1．软件管理 TLB

前面我们介绍的 TLB 管理和 TLB 故障的处理都完全由 MMU 硬件完成，只有一个页面不在内存时才会陷入操作系统。

而实际上，在现代的一些 RISC 机中，包括 MIPS、Alpha、HP PA，几乎全部的这种页面管理工作都是由软件完成的。在这些机器中，TLB 条目是由操作系统显式地装入，在 TLB 没有命中时，MMU 不是到页表中找到并装入需要的页面信息，而是产生一个 TLB 故障把问题交给操作系统。操作系统必须找到页面，从 TLB 中淘汰一个条目，装入一个新的条目，然后重新启动产生异常（或故障）的指令。当然，所有这些都必须用很少指令完成，因为 TLB 不命中的频率远比页面异常大得多。

令人惊奇的是，如果 TLB 的尺寸取一个合理的较大值(比如 64 个条目)以减少不命中的频率，那么软件管理的 TLB 效率可能相当高。这里主要的收益是一个简单得多的 MMU（最后介绍），它在 CPU 芯片上为高速缓存和其他能提高性能的部件让出了相当大的面积。

人们已经使用了很多方法来提高使用软件管理 TLB 机器的性能,有一个方法既能减少 TLB 的不命中率又能减少在 TLB 不命中确实发生时的开销。为了减少 TLB 的不命中率，操作系统有时可以用它的直觉来指出那些页面可能将被使用并把他们预装入 TLB 中。例如，当一个客户进程向位于同一台机器的服务器进程发出一个 RPC 请求时，服务器很可能即将运行。知道了这一点，在客户进程因执行 RPC 陷入时，系统就可以找到服务器的代码、数据、堆栈的页面，并在 TLB 中提前为他们建立映射，以避免 TLB 故障的发生。

无论是硬件还是软件，处理 TLB 不命中的一般方法是对页表执行索引操作找出所引用的页面。用软件执行这个搜索的一个问题是保存页表的页面本身可能就不在 TLB 中，这将在处理过程中再一次引发一个 TLB 异常,这种异常可以通过保持一个大的(比如 4KB)TLB 条目的软件高速缓存而得到减少，这个高速缓存保持在固定位置，它的页面总是保持在 TLB 中，操作系统通过首先检查软件高速缓存可以大大减少 TLB 不命中的次数。

2．刷新机制

用软件来管理 TLB 和其他缓存的一个重要的要求就是保持 TLB 和其他缓存中的内容的同步性，这样必须考虑在一定条件下刷新内容。

在 Linux 中刷新机制(包括 TLB 的刷新，缓存的刷新等等）主要要用来完成以下几个工作：

（1）保证在任何时刻内存管理硬件所看到的进程的内核映射和内核页表一致；

（2）如果负责内存管理的内核代码对用户进程页面进行了修改，那么用户的进程在被允许继续执行前,要求必须在缓存中看到正确的数据。

例如当正在执行 write() 系统调用时，要保证页面缓存中的页面为新页，也就是要使缓存中的页面内容和写入文件的一致，就需要更新缓存中的页面。

3．通常当地址空间的状态改变时，调用适当的刷新机制来描述状态的改变

在 Linux 中刷新机制的实现是通过一系列函数（或宏）来完成的,例如常用的两个刷新函数的一般形式为：

```
flush_cache_foo（ ）;
flush_tlb_foo（ ）;
```

这两个函数的调用是有一定顺序的，它们的逻辑意义如下所述。

在地址空间改变前必须刷新缓存，防止缓存中存在非法的空映射。函数 flush_cache_*() 会把缓存中的映射变成无效（ 这里的缓存指的是 MMU 中的缓存，它负责虚地址到物理地址的当前映射关系;注意在这里由于各种处理器中 MMU 的内部结构不同,换存刷新函数也不尽相同。比如在 80386 处理器中这些函数是为空——i386 处理器刷新时不需要任何多余的 MMU 的信息，内核页表包含了所有的必要信息）。在刷新地址后，由于页表的改变，必须刷新 TBL 以便硬件可以把新的页表信息装入 TLB。

下面介绍一些刷新函数的作用和使用情况：

```
void flush_cache_all（void）;
void flush_tlb_all（void）;
```

这两个例程是用来通知相应机制，内核地址空间的映射已被改变，它意味着所有的进程都被改变了；

```
void flush_cache_mm（struct mm_struct *mm）;
void flush_tlb_mm（struct mm_struct *mm）;
```

它们用来通知系统被 mm_struct 结构所描述的地址空间正在改变，它们仅发生在用户空间的地址改变时；

```
flush_cache_range（struct mm_struct *mm,unsigned long start, unsigned long end）;
flush_tlb_range（struct mm_struct *mm,unsigned long start, unsigned long end）;
```

它们刷新用户空间中的指定范围；

```
void flush_cache_page（struct vm_area_struct *vma,unsigned long address）;
void flush_tlb_page（struct vm_area_struct *vma,unsigned long address）;
```

刷新一页面。

void flush_page_to_ram(unsigned long page);/*如果使用 i386 处理器，此函数为空，相应的刷新功能由硬件内部自动完成*/

这个函数一般用在写时复制，它会使虚拟缓存中的对应项无效，这是因为如果虚拟缓存

不可以自动地回写，于是会造成虚拟缓存中页面和主存中的内容不一致。

例如,如图 6.23 所示，虚拟内存 0x2000 对任务 1、任务 2、任务 3 共享，但对任务 2 只是可读，它映射物理内存 0x1000,那么如果任务 2 要对虚拟内存 0x2000 执行写操作时，会产生页面错误。内存管理系统要给它重新分配一个物理页面如 0x2600,此页面的内容是物理内存 0x1000 的拷贝，这时虚拟索引缓存中就有两项内核别名项 0x2000 分别对应两个物理地址 0x1000 和 0x2600，在任务 2 对物理页面 0x2600 的内容进行了修改后，这样内核别名即虚地址 0x2000 映射的物理页面内容不一致，任务 3 在来访问虚地址 0x2000 时就会产生不一致错误。为了避免不一致错误，使用 flush_page_to_ram 使得缓存中的内核别名无效。

一般刷新函数的使用顺序如下：

```
copy_cow_page（old_page,new_page,address）;
flush_page_to_ram（old_page）;
flush_page_to_ram（new_page）;
flush_cache_page（vam,address）;
    ….
free_page（old_page）;
flush_tlb_page（vma,address）;
```

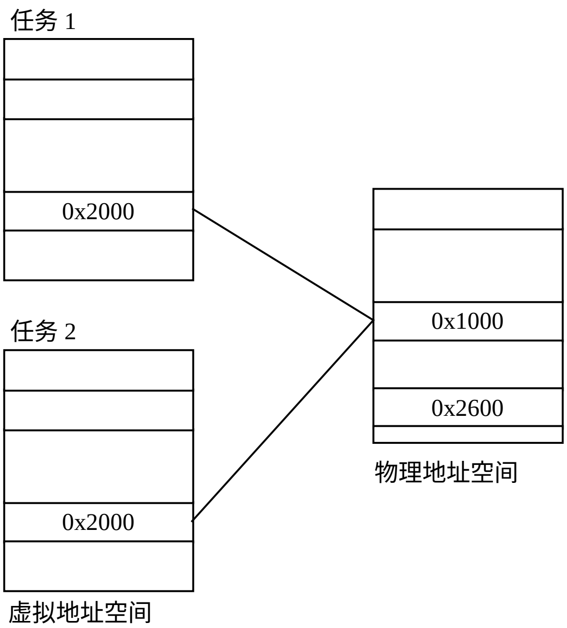


图 6.23　多个任务共享内存空间

4．函数代码简介

大部分刷新函数都在 include/asm/pttable.h 中定义，这里就 i386 中__flush_tlb()的定义给予说明：

```
#define __flush_tlb ()
     do {
            unsigned int tmpreg;
```

```
                __asm__ __volatile__ (
                        "movl %%cr3, %0;  # flush TLB \n"
                        "movl %0, %%cr3;              \n"
                        : "=r"  (tmpreg)
                        :: "memory") ;
        } while  (0)
```

这个函数比较简单，通过对 CR3 寄存的重新装入，完成对 TLB 的刷新。

6.8 进程的创建和执行

6.8.1 进程的创建

新的进程通过克隆旧的程序（当前进程）而建立。fork() 和 clone()（对于线程）系统调用可用来建立新的进程。这两个系统调用结束时，内核在系统的物理内存中为新的进程分配新的 task_struct 结构，同时为新进程要使用的堆栈分配物理页。Linux 还会为新的进程分配新的进程标识符。然后，新 task_struct 结构的地址保存在链表中，而旧进程的 task_struct 结构内容被复制到新进程的 task_struct 结构中。

在克隆进程时，Linux 允许两个进程共享相同的资源。可共享的资源包括文件、信号处理程序和虚拟内存等（通过继承）。当某个资源被共享时，该资源的引用计数值会增加 1，从而只有两个进程均终止时，内核才会释放这些资源。图 6.24 说明了父进程和子进程共享打开的文件。

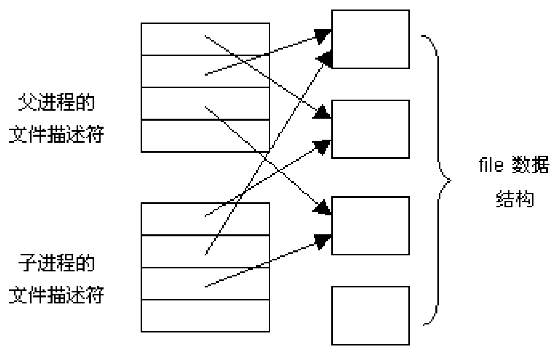


图 6.24　父进程和子进程共享打开的文件

系统对进程虚拟内存的克隆过程则更加巧妙些。新的 vm_area_struct 结构、新进程自己的 mm_struct 结构以及新进程的页表必须在一开始就准备好，但这时并不复制任何虚拟内存。如果旧进程的某些虚拟内存在物理内存中，而有些在交换文件中，那么虚拟内存的复制将会非常困难和费时。实际上，Linux 采用了称为写时复制的技术，也就是说，只有当两个进程中的任意一个向虚拟内存中写入数据时才复制相应的虚拟内存；而没有写入的任何内存

页均可以在两个进程之间共享。代码页实际总是可以共享的。

此外,内核线程是调用 kernel_thread()函数创建的,而 kernel_thread()在内核态调用了 clone()系统调用。内核线程通常没有用户地址空间，即 p->mm = NULL，它总是直接访问内核地址空间。

不管是 fork()还是 clone（）系统调用，最终都调用了内核中的 do_fork()，其源代码在 kernel/fork.c：

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information  (task[nr])  and sets up the necessary registers. It also
 * copies the data segment in its entirety.  The "stack_start" and
 * "stack_top" arguments are simply passed along to the platform
 * specific copy_thread ()  routine.  Most platforms ignore stack_top.
 * For an example that's using stack_top, see
 * arch/ia64/kernel/process.c.
 */
int do_fork (unsigned long clone_flags, unsigned long stack_start,
             struct pt_regs *regs, unsigned long stack_size)
{
        int retval;
        struct task_struct *p;
        struct completion vfork;

        retval = -EPERM;

        /*
         * CLONE_PID is only allowed for the initial SMP swapper
         * calls
         */
        if  (clone_flags & CLONE_PID)  {
                if  (current->pid)
                        goto fork_out;
        }

        retval = -ENOMEM;
        p = alloc_task_struct () ;
        if  (!p)
                goto fork_out;

        *p = *current;

        retval = -EAGAIN;
        /*
         * Check if we are over our maximum process limit, but be sure to
         * exclude root. This is needed to make it possible for login and
         * friends to set the per-user process limit to something lower
         * than the amount of processes root is running. -- Rik
         */
    if  (atomic_read (&p->user->processes)  >= p->rlim[RLIMIT_NPROC].rlim_cur
                  && !capable (CAP_SYS_ADMIN)  !capable (CAP_SYS_RESOURCE) )
                goto bad_fork_free;
```

```
	atomic_inc (&p->user->__count) ;
	atomic_inc (&p->user->processes) ;

	/*
	 * Counter increases are protected by
	 * the kernel lock so nr_threads can't
	 * increase under us  (but it may decrease) .
	 */
	if  (nr_threads >= max_threads)
		goto bad_fork_cleanup_count;

	get_exec_domain (p->exec_domain) ;

	if  (p->binfmt && p->binfmt->module)
		__MOD_INC_USE_COUNT (p->binfmt->module) ;

	p->did_exec = 0;
	p->swappable = 0;
	p->state = TASK_UNINTERRUPTIBLE;

	copy_flags (clone_flags, p) ;
	p->pid = get_pid (clone_flags) ;

	p->run_list.next = NULL;
	p->run_list.prev = NULL;

	p->p_cptr = NULL;
	init_waitqueue_head (&p->wait_chldexit) ;
	p->vfork_done = NULL;
	if  (clone_flags & CLONE_VFORK)  {
		p->vfork_done = &vfork;
		init_completion (&vfork) ;
	}
	spin_lock_init (&p->alloc_lock) ;

	p->sigpending = 0;
	init_sigpending (&p->pending) ;

	p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
	p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
	init_timer (&p->real_timer) ;
	p->real_timer.data =  (unsigned long)  p;

	p->leader = 0;		/* session leadership doesn't inherit */
	p->tty_old_pgrp = 0;
	p->times.tms_utime = p->times.tms_stime = 0;
	p->times.tms_cutime = p->times.tms_cstime = 0;
#ifdef CONFIG_SMP
	{
		int i;
		p->cpus_runnable = ~0UL;
```

```
                p->processor = current->processor;
                /* ?? should we just memset this ?? */
                for (i = 0; i < smp_num_cpus; i++)
                        p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
                spin_lock_init (&p->sigmask_lock) ;
        }
#endif
        p->lock_depth = -1;             /* -1 = no lock */
        p->start_time = jiffies;

        INIT_LIST_HEAD (&p->local_pages) ;

        retval = -ENOMEM;
        /* copy all the process information */
        if (copy_files (clone_flags, p) )
                goto bad_fork_cleanup;
        if (copy_fs (clone_flags, p) )
                goto bad_fork_cleanup_files;
        if (copy_sighand (clone_flags, p) )
                goto bad_fork_cleanup_fs;
        if (copy_mm (clone_flags, p) )
                goto bad_fork_cleanup_sighand;
        retval = copy_thread (0, clone_flags, stack_start, stack_size, p, regs) ;
        if (retval)
                goto bad_fork_cleanup_mm;
        p->semundo = NULL;

        /* Our parent execution domain becomes current domain
           These must match for thread signalling to apply */

        p->parent_exec_id = p->self_exec_id;

        /* ok, now we should be set up.. */
        p->swappable = 1;
        p->exit_signal = clone_flags & CSIGNAL;
        p->pdeath_signal = 0;

        /*
         * "share" dynamic priority between parent and child, thus the
         * total amount of dynamic priorities in the system doesnt change,
         * more scheduling fairness. This is only important in the first
         * timeslice, on the long run the scheduling behaviour is unchanged.
         */
        p->counter = (current->counter + 1) >> 1;
        current->counter >>= 1;
        if (!current->counter)
                current->need_resched = 1;

        /*
         * Ok, add it to the run-queues and make it
         * visible to the rest of the system.
         *
```

```
         * Let it rip!
         */
         retval = p->pid;
        p->tgid = retval;
        INIT_LIST_HEAD (&p->thread_group) ;

        /* Need tasklist lock for parent etc handling! */
         write_lock_irq (&tasklist_lock) ;

        /* CLONE_PARENT and CLONE_THREAD re-use the old parent */
        p->p_opptr = current->p_opptr;
         p->p_pptr = current->p_pptr;
         if  (! (clone_flags &  (CLONE_PARENT | CLONE_THREAD) ) )  {
                p->p_opptr = current;
                 if  (! (p->ptrace & PT_PTRACED) )
                         p->p_pptr = current;
         }

        if  (clone_flags & CLONE_THREAD)  {
                 p->tgid = current->tgid;
                 list_add (&p->thread_group, &current->thread_group) ;
        }

         SET_LINKS (p) ;
         hash_pid (p) ;
        nr_threads++;
         write_unlock_irq (&tasklist_lock) ;

        if  (p->ptrace & PT_PTRACED)
                 send_sig (SIGSTOP, p, 1) ;

         wake_up_process (p) ;             /* do this last */
        ++total_forks;
         if  (clone_flags & CLONE_VFORK)
                 wait_for_completion (&vfork) ;

fork_out:
         return retval;

bad_fork_cleanup_mm:
         exit_mm (p) ;
bad_fork_cleanup_sighand:
         exit_sighand (p) ;
bad_fork_cleanup_fs:
         exit_fs (p) ; /* blocking */
bad_fork_cleanup_files:
         exit_files (p) ; /* blocking */
bad_fork_cleanup:
         put_exec_domain (p->exec_domain) ;
        if  (p->binfmt && p->binfmt->module)
                 __MOD_DEC_USE_COUNT (p->binfmt->module) ;
bad_fork_cleanup_count:
```

```
            atomic_dec (&p->user->processes) ;
            free_uid (p->user) ;
    bad_fork_free:
            free_task_struct (p) ;
         goto fork_out;
    }
```

尽管 fork () 系统调用因为传递用户堆栈和寄存器参数而与特定的平台相关，但实际上 do_fork()所做的工作还是可移植的。下面给出对以上代码的解释。

给局部变量赋初值-ENOMEM,当分配一个新的 task_struc 结构失败时就返回这个错误值。

如果在 clone_flags 中设置了 CLONE_PID 标志，就返回一个错误（-EPERM）。因为 CLONE_PID 有特殊的作用，当这个标志为 1 时，父、子进程（线程）共用一个进程号，也就是说，子进程虽然有自己的 task_struct 结构，却使用父进程的 pid。但是，只有 0 号进程（即系统中的空线程）才允许使用这个标志。

调用 alloc_task_struct()为子进程分配两个连续的物理页面，低端用来存放子进程的 task_struct 结构，高端用作其内核空间的堆栈。

用结构赋值语句*p = *current 把当前进程 task_struct 结构中的所有内容都拷贝到新进程中。稍后，子进程不该继承的域会被设置成正确的值。

在 task_struct 结构中有个指针 user，用来指向一个 user_struct 结构。一个用户常常有多个进程，所以有关用户的信息并不专属于某一个进程。这样，属于同一用户的进程就可以通过指针 user 共享这些信息。显然，每个用户有且只有一个 user_struct 结构。该结构中有一个引用计数器 count，对属于该用户的进程数量进行计数。可想而知，内核线程并不属于某个用户，所以其 task_struct 中的 user 指针为 0。每个进程 task_struct 结构中有个数组 rlim，对该进程占用各种资源的数量作出限制，而 rlim[RLIMIT_NPROC]就规定了该进程所属用户可以拥有的进程数量。所以，如果当前进程是一个用户进程，并且该用户拥有的进程数量已经达到了规定的界限值，就不允许它 fork()了。

除了检查每个用户拥有的进程数量外，接着要检查系统中的任务总数（所有用户的进程数加系统的内核线程数）是否超过了最大值 max_threads，如果是，也不允许再创建子进程。

一个进程除了属于某个用户外，还属于某个“执行域”。Linux 可以运行 X86 平台上其他 UNIX 类操作系统生成的符合 iBCS2 标准的程序。例如，一个进程所执行的程序是为 Solaris 开发的，那么这个进程就属于 Solaris 执行域 PER_SOLARIS。当然，在 Linux 上运行的绝大多数程序属于 Linux 执行域。在 task_struct 有一个指针 exec_doman，指向一个 exec_doman 结构。在 exec_doman 结构中有一个域是 module，这是指向某个 module 结构的指针。在 Linux 中，一个文件系统或驱动程序都可以作为一个单独的模块进行编译，并动态地链接到内核中。在 module 结构中有一个计数器 count，用来统计几个进程需要使用这个模块。因此，get_exec_domain(p->exec_domain)递增模块结构 module 中的计数器。

另外，每个进程所执行的程序属于某种可执行映像格式，如 a.out 格式、elf 格式，甚至 Java 虚拟机格式。对于不同格式的支持通常是通过动态安装的模块来实现的。所以，task_struct 中有一个执行 linux_binfmt 结构的指针 binfmt，而__MOD_INC_USE_COUNT () 就是对有关模块的使用计数进行递增。

紧接着为什么要把进程的状态设置成为 TASK_UNINTERRUPTIBLE？这是因为后面

get_pid()的操作必须独占，子进程可能因为一时进不了临界区而只好暂时进入睡眠状态。

copy_flags()函数将clone_flags参数中的标志位略加补充和变换,然后写入p->flags。

get_pid () 函数根据 clone_flags 中标志位 ClONE_PID 的值，或返回父进程(当前进程)的 pid，或返回一个新的 pid。

前面在复制父进程的 task_struct 结构时把父进程的所有域都照抄过来，但实际上很多域的值必须重新赋初值，因此，后面的赋值语句就是对子进程 task_struct 结构的初始化。其中 start_time 表示进程创建的时间,而全局变量 jiffies 就是从系统初始化开始至当前的是时钟滴答数。local_pages 表示属于该进程的局部页面形成一个双向链表，在此进行了初始化。

copy_files()有条件地复制已打开文件的控制结构，也就是说，这种复制只有在 clone_flags 中的 CLONE_FILES 标志为 0 时才真正进行，否则只是共享父进程的已打开文件。当一个进程有已打开文件时，task_struct 结构中的指针 files 指向一个 fiel_struct 结构，否则为 0。所有与终端设备 tty 相联系的用户进程的头 3 个标准文件 stdin、stdout 及 stderr 都是预先打开的，所以指针一般不为空。

copy_fs()也是只有在 clone_flags 中的 CLONE_FS 标志为 0 时才加以复制。在 task_struct 中有一个指向 fs_struct 结构的指针,fs_struct 结构中存放的是进程的根目录 root、当前工作目录 pwd、一个用于文件操作权限的 umask，还有一个计数器。类似地，copy_sighand（）也是只有在 CLONE_SIGHAND 为 0 时才真正复制父进程的信号结构，否则就共享父进程。信号是进程间通信的一种手段，信号随时都可以发向一个进程，就像中断随时都可以发向一个处理器一样。进程可以为各种信号设置相应的信号处理程序，一旦进程设置了信号处理程序，其 task_struct 结构中的指针 sig 就指向 signal_struct 结构（定义于 include/linux/sched.h）。关于信号的具体内容将在下一章进行介绍。

用户空间的继承是通过 copy_mm()函数完成的。进程的 task_struct 结构中有一个指针 mm,就指向代表着进程地址空间的 mm_struct 结构。对 mm_struct 的复制也是在 clone_flags 中的 CLONE_VM 标志为 0 时才真正进行，否则，就只是通过已经复制的指针共享父进程的用户空间。对 mm_struct 的复制不只限于这个数据结构本身，还包括了对更深层次数据结构的复制，其中最主要的是 vm_area_struct 结构和页表的复制，这是由同一文件中的 dum_mmap()函数完成的。

到此为止，task_struct 结构中的域基本复制好了，但是用于内核堆栈的内容还没有复制，这就是 copy_thread（）的责任了。copy_thread（）函数与平台相关，定义于 arch/i386/kernel/process.c 中。copy_thread（）实际上只复制父进程的内核空间堆栈。堆栈中的内容记录了父进程通过系统调用 fork()进入内核空间、然后又进入 copy_thread()函数的整个历程，子进程将要循相同的路线返回，所以要把它复制给子进程。但是，如果父子进程的内核空间堆栈完全相同，那返回用户空间后就无法区分哪个是子进程了，所以，复制以后还要略作调整。有兴趣的读者可以结合第三、四章内容去读该函数的源代码。

parent_exec_id 表示父进程的执行域，p->self_exec_id 是本进程（子进程）的执行域，swappable 表示本进程的页面可以被换出。exit_signal 为本进程执行 exit()系统调用时向父进程发出的信号，pdeath_signal 为要求父进程在执行 exit（）时向本进程发出的信号。另外，counter 域的值是进程的时间片（以时钟滴达为单位），代码中将父进程的时间片分

成两半，让父、子进程各有原值的一半。

进程创建后必须处于某一组中，这是通过 task_struct 结构中的队列头 thread_group 与父进程链接起来，形成一个进程组（注意，thread 并不单指线程，内核代码中经常用 thread 通指所有的进程）。

建立进程的家族关系。先建立起子进程的祖先和双亲（当然还没有兄弟和孩子），然后通过 SET_LINKS（）宏将子进程的 task_struct 结构插入到内核中其他进程组成的双向链表中。通过 hash_pid()将其链入按其 pid 计算得的哈希表中（参看第四章进程组织方式一节）。

最后，通过 wake_up_process()将子进程唤醒，也就是将其挂入可执行队列等待被调度。

但是，还有一种特殊情况必须考虑。当参数 clone_flags 中 CLONE_VFORK 标志位为 1 时，一定要保证子进程先运行，一直到子进程通过系统调用 execve()执行一个新的可执行程序或通过系统调用 exit()退出系统时，才可以恢复父进程的执行，这是通过 wait_for_completion（）函数实现的。为什么要这样做呢？这是因为当 CLONE_VFORK 标志位为 1 时，就说明父、子进程通过指针共享用户空间（指向相同的 mm_struct 结构），那也说明父进程写入用户空间的内容同时也写入了子进程的用户空间，反之亦然。如果说，在这种情况下，父子进程对数据区的写入可能引起问题的话，那么，对堆栈区的写入可能就是致命的了。而对子程序或函数的调用肯定就是对堆栈的写入。由此可见，在这种情况下，决不能让两个进程都回到用户空间并发执行，否则，必然导致两个进程的互相“捣乱”或因非法访问而死亡。解决的办法的只能是“扣留”其中的一个进程，而让另一个进程先回到用户空间，直到两个进程不再共享它们的用户空间，或其中一个进程消亡为止（肯定是先回到用户空间的进程先消亡）。

到此为止，子进程的创建已经完成，该是从内核态返回用户态的时候了。实际上，fork()系统调用执行之后，父子进程返回到用户空间中相同的的地址，用户进程根据 fork()的返回值分别安排父子进程执行不同的代码。

6.8.2　程序执行

与 UNIX 类似，Linux 中的程序和命令通常由命令解释器执行，这一命令解释器称为 shell。用户输入命令之后，shell 会在搜索路径（shell 变量 PATH 中包含搜索路径）指定的目录中搜索和输入命令匹配的映像（可执行的二进制代码）名称。如果发现匹配的映像，shell 负责装载并执行该映像。shell 首先利用 fork 系统调用建立子进程，然后用找到的可执行映像文件覆盖子进程正在执行的 shell 二进制映像。

可执行文件可以是具有不同格式的二进制文件，也可以是一个文本的脚本文件。可执行映像文件中包含了可执行代码及数据，同时也包含操作系统用来将映像正确装入内存并执行的信息。Linux 使用的最常见的可执行文件格式是 ELF 和 a.out，但理论上讲，Linux 有足够的灵活性可以装入任何格式的可执行文件。

1．ELF 可执行文件

ELF 是“可执行可连接格式”的英文缩写，该格式由 UNIX 系统实验室制定。它是 Linux 中最经常使用的格式，和其他格式（例如 a.out 或 ECOFF 格式）比较起来，ELF 在装入内存时多一些系统开支，但是更为灵活。ELF 可执行文件包含了可执行代码和数据，通常也称为正文和数据。这种文件中包含一些表，根据这些表中的信息，内核可组织进程的虚拟内存。另外，文件中还包含有对内存布局的定义以及起始执行的指令位置。

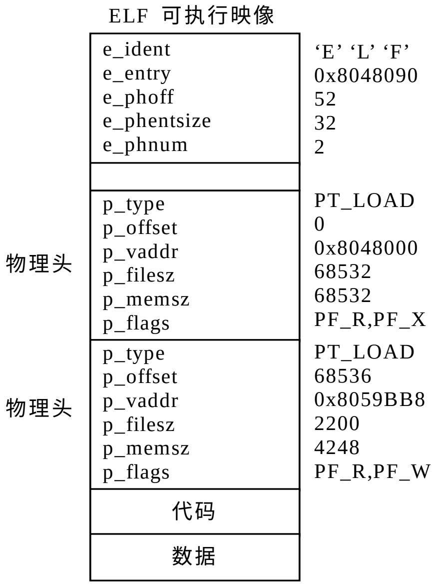


图 6.25　一个简单的 ELF 可执行文件的布局

下面我们分析一个简单程序在利用编译器编译并连接之后的 ELF 文件格式：

```
#include <stdio.h>

main  ()
{
printf（“Hello world!\n”）;

}
```

图 6_25 所示，是上述源代码在编译连接后的 ELF 可执行文件的格式。从图 可以看出，ELF 可执行映象文件的开头是 3 个字符‘E’、‘L’和‘F’，作为这类文件的标识符。*e_entry* 定义了程序装入之后起始执行指令的虚拟地址。这个简单的 ELF 映像利用两个“物理头”结构分别定义代码和数据，*e_phnum* 是该文件中所包含的物理头信息个数，本例为 2。*e_phyoff* 是第一个物理头结构在文件中的偏移量，而 *e_phentsize* 则是物理头结构的大小，这两个偏移量均从文件头开始算起。根据上述两个信息，内核可正确读取两个物理头结构中的信息。

物理头结构的 p_flags 字段定义了对应代码或数据的访问属性。图中第 1 个 p_flags 字段的值为 FP_X 和 FP_R，表明该结构定义的是程序的代码；类似地，第 2 个物理头定义程序数据，并且是可读可写的。p_offset 定义对应的代码或数据在物理头之后的偏移量。p_vaddr 定义代码或数据的起始虚拟地址。p_filesz 和 p_memsz 分别定义代码或数据在文件中的大小以及在内存中的大小。对我们的简单例子，程序代码开始于两个物理头之后，而程序数据则开始于物理头之后的第 0x68533 字节处，显然，程序数据紧跟在程序代码之后。程序的代码大小为 0x68532，显得比较大，这是因为连接程序将 C 函数 printf 的代码连接到了 ELF 文件的原因。程序代码的文件大小和内存大小是一样的，而程序数据的文件大小和内存大小不一样，这是因为内存数据中，起始的 2200 字节是预先初始化的数据，初始化值来自 ELF 映象，而其后的 2048 字节则由执行代码初始化。

如前面所描述的，Linux 利用请页技术装入程序映像。当 shell 进程利用 fork ()系统调用建立了子进程之后，子进程会调用 exec ()系统调用（实际有多种 exec 调用），exec()系统调用将利用 ELF 二进制格式装载器装载 ELF 映像，当装载器检验映像是有效的 ELF 文件之后，就会将当前进程（实际就是父进程或旧进程）的可执行映像从虚拟内存中清除，同时清除任何信号处理程序并关闭所有打开的文件（把相应 file 结构中的 f_count 引用计数

减 1，如果这一计数为 0，内核负责释放这一文件对象），然后重置进程页表。完成上述过程之后，只需根据 ELF 文件中的信息将映象代码和数据的起始和终止地址分配并设置相应的虚拟地址区域，修改进程页表。这时，当前进程就可以开始执行对应的 ELF 映像中的指令了。

2．命令行参数和 shell 环境

当用户敲入一个命令时，从 shell 可以接受一些命令行参数。例如，当用户敲入命令：

```
$ ls -l /usr/bin
```

以获得在/usr/bin 目录下的全部文件列表时，shell 进程创建一个新进程执行这个命令。这个新进程装入/bin/ls 可执行文件。在这样做的过程中，从 shell 继承的大多数执行上下文被丢弃，但 3 个单独的参数 ls、-l 和 /usr/依然被保持。一般情况下，新进程可以接受任意个参数。

传递命令行参数的约定依赖于所用的高级语言。在 C 语言中，程序的 main( )函数把传递给程序的参数个数和指向字符串指针数组的地址作为参数。下面是 main()的原型：

```
int main (int argc, char *argv[])
```

再回到前面的例子，当/bin/ls 程序被调用时，argc 的值为 3，argv[0]指向 ls 字符串，argv[1]指向-l 字符串，而 argv[2]指向 /usr/bin 字符串。argv 数组的末尾处总以空指针来标记，因此，argv[3]为 NULL。

在 C 语言中传递给 main()函数的第 3 个可选参数是包含环境变量的参数。当进程用到它时，main( )的声明如下：

```
int main (int argc, char *argv[], char *envp[])
```

envp 参数指向环境串的指针数组，形式如下：

```
VAR_NAME=something
```

在这里，VAR_NAME 表示一个环境变量的名字，而“=”后面的子串表示赋给变量的实际值。envp 数组的结尾用一个空指针标记，就像 argv 数组。环境变量是用来定制进程的执行上下文，为用户或其他进程提供一般的信息，或允许进程交叉调用 execve( )系统调用保存一些信息。

命令行参数和环境串都放在用户态堆栈。图 6.26 显示了用户态堆栈底部所包含的内容。注意环境变量位于栈底附近正好在一个 NULL 的长整数之后。

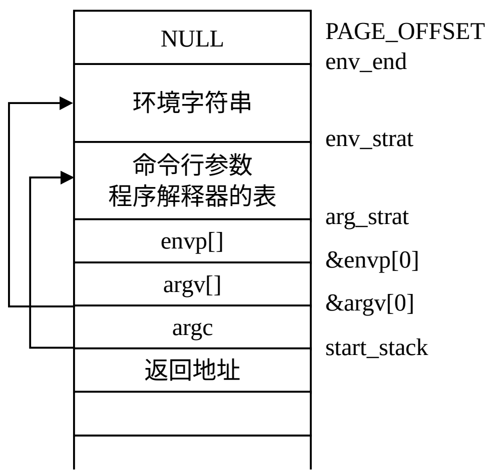


图 6.26　用户态堆栈底部所包含的内容

3．函数库

每个高级语言的源代码文件都是经过几个步骤才转化为目标文件的，目标文件中包含的是汇编语言指令的机器代码，它们和相应的高级语言指令对应。目标文件并不能被执行，因为它不包含源代码文件所用的全局外部符号名的虚拟地址。这些地址的分配或解析是由链接程序完成的，链接程序把程序所有的目标文件收集起来并构造可执行文件。链接程序还分析程序所用的库函数并把它们粘合成可执行文件。

任何程序，甚至最小的程序都会利用 C 库。请看下面的一行 C 程序：

```
void main (void) { }
```

尽管这个程序没有做任何事情，但还是需要做很多工作来建立执行环境并在程序终止时杀死这个进程。尤其是，当 main( )函数终止时，C 编译程序就把 exit( )系统调用插入到目标代码中。

实际上，一般程序对系统调用的调用通常是通过 C 库中的封装例程进行的，也就是说，C 语言函数库中的函数先调用系统调用，而我们的应用程序再调用库函数。除了 C 库，UNIX 系统中还包含很多其他的函数库。一般的 Linux 系统可能轻而易举地就有 50 个不同的库。这里仅仅列举其中的两个：数学库 libm 包含浮点操作的基本函数，而 X11 库 libX11 收集了所有 X11 窗口系统图形接口的基本底层函数。

传统 UNIX 系统中的所有可执行文件都是基于静态库的。这就意味着链接程序所产生的可执行文件不仅包括源程序的代码，还包括程序所引用的库函数的代码。

静态库的一大缺点是：它们占用大量的磁盘空间。的确，每个静态链接的可执行文件都复制库代码的一部分。因此，现代 UNIX 系统利用了共享库。可执行文件不用再包含库的目标代码，而仅仅指向库名。当程序被装入内存执行时，一个叫做程序解释器的程序就专注于分析可执行文件中的库名，确定所需库在系统目录树中的位置，并使执行进程可以使用所请求的代码。

共享库对提供文件内存映射的系统尤为方便，因为它们减少了执行一个程序所需的主内存量。当程序解释器必须把某一共享库链接到进程时，并不拷贝目标代码，而是仅仅执行一个内存映射，把库文件的相关部分映射到进程的地址空间中。这就允许共享库机器代码所在的页面由使用相同代码的所有进程进行共享。

共享库也有一些缺点。动态链接的程序启动时间通常比静态链接的长。此外，动态链接的程序的可移植性也不如静态链接的好，因为当系统中所包含的库版本发生变化时，动态链接的程序可能就不能适当地执行。

用户可以让一个程序静态地链接。例如，GCC 编译器提供-static 选项，即告诉链接程序使用静态库而不是共享库。

和静态连接库不同，动态连接库只有在运行时才被连接到进程的虚拟地址中。对于使用同一动态连接库的多个进程，只需在内存中保留一份共享库信息即可，这样就节省了内存空间。当共享库需要在运行时连接到进程虚拟地址时，Linux 的动态连接器利用 ELF 共享库中的符号表完成连接工作，符号表中定义了 ELF 映像引用的全部动态库例程。Linux 的动态连

接器一般包含在 /lib 目录中，通常为 ld.so.1、llibc.so.1 和 ld-linux.so.1。

6.8.3　执行函数

在执行 fork()之后，同一进程有两个拷贝都在运行，也就是说，子进程具有与父进程相同的可执行程序和数据（简称映像）。但是，子进程肯定不满足于仅仅成为父进程的“影子”，因此，父进程就要调用 execve()装入并执行子进程自己的映像。execve()函数必须定位可执行文件的映像，然后装入并运行它。当然开始装入的并不是实际二进制映像的完全拷贝，拷贝的完全装入是用请页装入机制（Demand Pageing Loading）逐步完成的。开始时只需要把要执行的二进制映像头装入内存，可执行代码的 inode 节点被装入当前进程的执行域中就可以执行了。

由于 Linux 文件系统采用了 linux_binfmt 数据结构（在/include/linux/ binfmt.h 中，见文件系统注册）来支持各种文件系统，所以 Linux 中的 exec()函数执行时，使用已注册的 linux_binfmt 结构就可以支持不同的二进制格式，即多种文件系统（EXT2,dos 等）。需要指出的是 binux_binfmt 结构中嵌入了两个指向函数的指针，一个指针指向可执行代码，另一个指向了库函数；使用这两个指针是为了装入可执行代码和要使用的库。linux_binfmt 结构描述如下，其链表结构的示意图如图 6.27 所示。

```
struct linux_binfmt {
     struct linux_binfmt * next;
       long *use_count;
     int  (*load_binary) (struct linux_binprm *, struct  pt_regs * regs);/*装入二进制代码*/
     int  (*load_shlib) (int fd);   /*装入公用库*/
       int  (*core_dump) (long signr, struct pt_regs * regs);
```

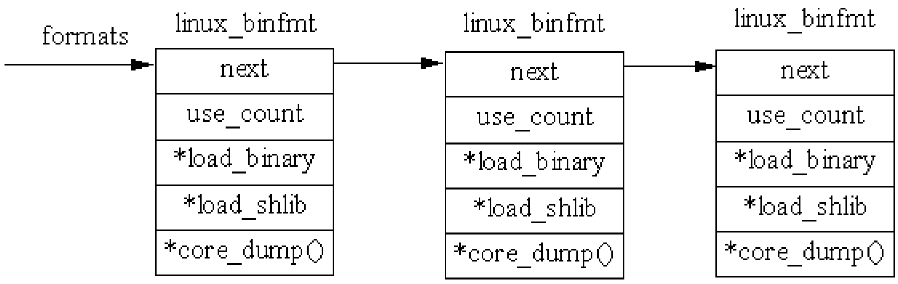


图 6.27　linux_binfmt 的链表结构

在使用这种数据结构前必须调用 vod binfmt_setup（）函数进行初始化；这个函数分别初始化了一些可执行的文件格式，如：init_elf_binfmt()；init_aout_binfmt()；init_java_binfmt()；init_script_binfmt())。

其实初始化就是用 register_binfmt(struct linux_binfmt * fmt)函数把文件格式注册到系统中，即加入*formats 所指的链中，*formats 的定义如下：

```
static struct linux_binfmt *formats = (struct linux_binfmt *) NULL
```

在使用装入函数的指针时，如果可执行文件是 ELF 格式的，则指针指向的装入函数分别

是：

```
load_elf_binary (struct linux_binprm * bprm, struct pt_regs * regs) ;
static int load_elf_library (int fd) ;
```

所以 elf_format 文件格式说明将被定义成：

```
static struct linux_binfmt elf_format = {#ifndef MODULE
    NULL, NULL, load_elf_binary, load_elf_library, elf_core_dump#else
    NULL, &mod_use_count_, load_elf_binary, load_elf_library, elf_core_dump#endif }
```

其他格式文件处理很类似，相关代码请看本节后面介绍的 search_binary_handler()函数。

另外还要提的是在装入二进制时还需要用到结构 linux_binprm，这个结构保存着一些在装入代码时需要的信息：

```
struct linux_binprm{
   char buf[128];/*读入文件时用的缓冲区*/
     unsigned long page[MAX_ARG_PAGES];
     unsigned long p;
     int sh_bang;
     struct inode * inode;/*映像来自的节点*/
     int e_uid, e_gid;
   int argc, envc; /*参数数目，环境数目*/
     char * filename;  /* 二进制映像的名字，也就是要执行的文件名 */
     unsigned long loader, exec;
     int dont_iput;        /* binfmt handler has put inode */
};
```

其他域的含义在后面的 do_exec()代码中做进一步解释。

Linux 所提供的系统调用名为 execve()，可是，C 语言的程序库在此系统调用的基础上向应用程序提供了一整套的库函数，包括 execve()、execlp()、execle()、execv()、execvp()，它们之间的差异仅仅是参数的不同。下面来介绍 execve()的实现。

系统调用 execve()在内核的入口为 sys_execve()，其代码在 arch/i386/kernel/process.c：

```
 /*
  * sys_execve ()  executes a new program.
  */
asmlinkage int sys_execve (struct pt_regs regs)
{
        int error;
        char * filename;

        filename = getname ( (char *)  regs.ebx) ;
        error = PTR_ERR (filename) ;
        if  (IS_ERR (filename) )
                goto out;
error = do_execve (filename,  (char **)  regs.ecx,  (char **)  regs.edx, &regs) ;
        if  (error == 0)
                current->ptrace &= ~PT_DTRACE;
        putname (filename) ;
out:
        return error;
}
```

系统调用进入内核时，regs.ebx 中的内容为应用程序中调用相应的库函数时的第 1 个参数，这个参数就是可执行文件的路径名。但是此时文件名实际上存放在用户空间中，所以 getname()要把这个文件名拷贝到内核空间，在内核空间中建立起一个副本。然后，调用 do_execve（）来完成该系统调用的主体工作。do_execve（）的代码在 fs/exec.c 中：

```
 /*
  * sys_execve ()  executes a new program.
  */
int do_execve (char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
         struct linux_binprm bprm;
         struct file *file;
         int retval;
         int i;

         file = open_exec (filename) ;

         retval = PTR_ERR (file) ;
         if  (IS_ERR (file) )
                 return retval;

        bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof (void *) ;
         memset (bprm.page, 0, MAX_ARG_PAGES*sizeof (bprm.page[0]) ) ;

         bprm.file = file;
         bprm.filename = filename;
         bprm.sh_bang = 0;
         bprm.loader = 0;
         bprm.exec = 0;
         if  ( (bprm.argc = count (argv, bprm.p / sizeof (void *) ) )  < 0)  {
                 allow_write_access (file) ;
                 fput (file) ;
                return bprm.argc;
              }

         if  ( (bprm.envc = count (envp, bprm.p / sizeof (void *) ) )  < 0)  {
                 allow_write_access (file) ;
                 fput (file) ;
                 return bprm.envc;
        }

         retval = prepare_binprm (&bprm) ;
         if  (retval < 0)
                 goto out;

        retval = copy_strings_kernel (1, &bprm.filename, &bprm) ;
        if  (retval < 0)
                 goto out;

         bprm.exec = bprm.p;
         retval = copy_strings (bprm.envc, envp, &bprm) ;
```

```
        if  (retval < 0)
                goto out;

        retval = copy_strings (bprm.argc, argv, &bprm) ;
         if  (retval < 0)
                goto out;

         retval = search_binary_handler (&bprm,regs) ;
         if  (retval >= 0)
                /* execve success */
                return retval;

out:
/* Something went wrong, return the inode and free the argument pages*/
         allow_write_access (bprm.file) ;
         if  (bprm.file)
                fput (bprm.file) ;

        for  (i = 0 ; i < MAX_ARG_PAGES ; i++)  {
                struct page * page = bprm.page[i];
               if  (page)
                      __free_page (page) ;
         }

        return retval;
}
```

参数 filename、argv、envp 分别代表要执行文件的文件名、命令行参数及环境串。下面对以上代码给予解释。

首先，将给定可执行程序的文件找到并打开，这是由 open_exec()函数完成的。open_exec()返回一个 file 结构指针，代表着所读入的可执行文件的映像。

与可执行文件路径名的处理办法一样，每个参数的最大长度也定为一个页面（是否有点浪费？），所有 linux_binprm 结构中有一个页面指针数组，数组的大小为系统所允许的最大参数个数 MAX_ARG_PAGES（定义为 32）。memset()函数将这个指针数组初始化为全 0。

对局部变量 bprm 的各个域进行初始化。其中 bprm.p 几乎等于最大参数个数所占用的空间； bprm.sh_bang 表示可执行文件的性质，当可执行文件是一个 Shell 脚本（Shell Sript）时置为 1，此时还没有可执行 Shell 脚本，因此给其赋初值 0，还有其他两个域也赋初值 0。

函数 count()对字符串数组 argv[]中参数的个数进行计数。bprm.p / sizeof(void *)表示所允许参数的最大值。同样，对环境变量也要统计其个数。

如果 count()小于 0，说明统计失败，则调用 fput()把该可执行文件写回磁盘，在写之前，调用 allow_write_access（）来防止其他进程通过内存映射改变该可执行文件的内容。

完成了对参数和环境变量的计数之后，又调用 prepare_binprm（）对 bprm 变量做进一步的准备工作。更具体地说，就是从可执行文件中读入开头的 128 个字节到 linux_binprm 结构的缓冲区 buf，这是为什么呢？因为不管目标文件是 ELF 格式还是 a.out 格式，或者其他格式，在其可执行文件的开头 128 个字节中都包括了可执行文件属性的信息，如图 6.25。

然后，就调用 copy_strings 把参数以及执行的环境从用户空间拷贝到内核空间的 bprm

变量中，而调用 copy_strings_kernel（）从内核空间中拷贝文件名，因为前面介绍的get_name()已经把文件名拷贝到内核空间了。

所有的准备工作已经完成，关键是调用 search_binary_handler（）函数了，请看下面对这个函数的详细介绍。

search_binary_handler（）函数也在 exec.c 中。其中有一段代码是专门针对 alpha 处理器的条件编译，在下面的代码中跳过了这段代码：

```
  /*
  * cycle the list of binary formats handler, until one recognizes the image
  */
int search_binary_handler (struct linux_binprm *bprm,struct pt_regs *regs)
{
         int try,retval=0;
         struct linux_binfmt *fmt;
6 #ifdef __alpha__
    …..
#endif
         /* kernel module loader fixup */
         /* so we don't try to load run modprobe in kernel space. */
         set_fs (USER_DS) ;
         for  (try=0; try<2; try++)  {
                read_lock (&binfmt_lock) ;
                 for  (fmt = formats ; fmt ; fmt = fmt->next)  {
                        int  (*fn)  (struct linux_binprm *, struct pt_regs *)  =
fmt->load_binary;
                       if  (!fn)
                               continue;
                        if  (!try_inc_mod_count (fmt->module) )
                               continue;
                        read_unlock (&binfmt_lock) ;
                        retval = fn (bprm, regs) ;
                       if  (retval >= 0)  {
                              put_binfmt (fmt) ;
                              allow_write_access (bprm->file) ;
                              if  (bprm->file)
                                     fput (bprm->file) ;
                              bprm->file = NULL;
                              current->did_exec = 1;
                              return retval;
                       }
                        read_lock (&binfmt_lock) ;
                        put_binfmt (fmt) ;
                        if  (retval != -ENOEXEC)
                              break;
                        if  (!bprm->file)  {
                               read_unlock (&binfmt_lock) ;
                              return retval;
                        }
                }
                 read_unlock (&binfmt_lock) ;
                if  (retval != -ENOEXEC)  {
```

```
                         break;
#ifdef CONFIG_KMOD
                }else{
#define printable (c)  ( ( (c) =='\t')  || ( (c) =='\n')  || (0x20<= (c)  && (c) <=0x7e) )
                         char modname[20];
                        if  (printable (bprm->buf[0])  &&
                             printable (bprm->buf[1])  &&
                             printable (bprm->buf[2])  &&
                            printable (bprm->buf[3]) )
                                break; /* -ENOEXEC */
                        sprintf  ( modname,  "binfmt-%04x",  *  ( unsigned  short  * )
(&bprm->buf[2]) ) ;
                        request_module (modname) ;
#endif
                  }
         }
          return retval;
}
```

在 exec.c 中定义了一个静态变量 formats:

static struct linux_binfmt *formats

因此，formats 就指向图 6.27 中链表队列的头，挂在这个队列中的成员代表着各种可执行文件格式。在 do_exec()函数的准备阶段,已经从可执行文件头部读入 128 字节存放在 bprm 的缓冲区中，而且运行所需的参数和环境变量也已收集在 bprm 中。search_binary_handler() 函数就是逐个扫描 formats 队列，直到找到一个匹配的可执行文件格式，运行的事就交给它。如果在这个队列中没有找到相应的可执行文件格式，就要根据文件头部的信息来查找是否有为此种格式设计的可动态安装的模块，如果有，就把这个模块安装进内核，并挂入 formats 队列，然后再重新扫描。下面对具体程序给予解释。

程序中有两层嵌套 for 循环。内层是针对 formats 队列的每个成员，让每一个成员都去执行一下 load_binary()函数，如果执行成功，load_binary()就把目标文件装入并投入运行，并返回一个正数或 0。当 CPU 从系统调用 execve()返回到用户程序时，该目标文件的执行就真正开始了，也就是，子进程新的主体真正开始执行了。如果 load_binary()返回一个负数，就说明或者在处理的过程中出错，或者没有找到相应的可执行文件格式，在后一种情况下，返回-ENOEXEC。

内层循环结束后，如果 load_binary()执行失败后的返回值为-ENOEXEC，就说明队列中所有成员都不认识目标文件的格式。这时，如果内核支持动态安装模块（取决于编译选项 CONFIG_KMOD），就根据目标文件的第 2 和第 3 个字节生成一个 binfmt 模块，通过 request_module()试着将相应的模块装入内核（参见第十章）。外层的 for 循环有两次，就是为了在安装了模块以后再来试一次。

在 linux_binfmt 数据结构中，有 3 个函数指针：load_binary、load_shlib 以及 core_dump，其中 load_binary 就是具体的装载程序。不同的可执行文件其装载函数也不同，如 a.out 格式的装载函数为 load_aout_binary(),ELF 格式的装载函数为 load_elf_binary(),其源代码分别在 fs/binfmt_aout.c 中和 fs/binfmt_elf 中。有兴趣的读者可以继续探究下去。

本章从内存的初始化开始，分别介绍了地址映射机制、内存分配与回收机制、请页机制、

交换机制、缓存和刷新机制、程序的创建及执行等 8 个方面。可以说，内存管理是整个操作系统中最复杂的一个子系统，因此，本章用大量的篇幅对相关内容进行了介绍，即使如此，也仅仅介绍了主要内容。

在本章的学习中，有一点需特别向读者强调。在 Linux 系统中，CPU 不能按物理地址访问存储空间，而必须使用虚拟地址。因此，对于 Linux 内核映像，即使系统启动时将其全部装入物理内存，也要将其映射到虚拟地址空间中的内核空间，而对于用户程序，其经过编译、链接后形成的映像文件最初存于磁盘，当该程序被运行时，先要建立该映像与虚拟地址空间的映射关系，当真正需要物理内存时，才建立地址空间与物理空间的映射关系。

第七章　进程间通信

进程为了能在同一项任务上协调工作，它们彼此之间必须能够进行通信。例如，在一个 shell 管道中，第 1 个进程的输出必须传送到第 2 个进程，这样沿着管道传递下去。因此在需要通信的进程之间，最好使用一种结构较好的通信方式。

Linux 支持许多不同形式的进程间通信机制 CIPCC，在特定的情况下，它们各自有优缺点。这一章将讨论最有用的进程间通信机制，即管道、 System V 的 IPC 机制及信号。至于 Linux 支持的完全网络兼容的进程间通信机制 Sockets，将在第十三章网络部分中介绍。

本章介绍的管道通信中，要讨论匿名管道和命名管道两种方式；而在 System V 的 IPC 机制中，要讨论信号量、消息队列及共享内存 3 种通信方式。对于信号，则主要讨论 Linux 的信号机制。

本章将从内核和系统调用两个角度来讨论这些通信机制，以使你在写协调工作的并发进程时，可以作出明智的选择。

7.1　管道

在进程之间通信的最简单的方法是通过一个文件，其中有一个进程写文件，而另一个进程从文件中读，这种方法比较简单，其优点体现在：

- 只要进程对该文件具有访问权限，那么，两个进程间就可以进行通信；
- 进程之间传递的数据量可以非常大。

尽管如此，使用文件进行进程间通信也有两大缺点。

- 空间的浪费。写进程只有确保把新数据加到文件的尾部，才能使读进程读到数据，对长时间存在的进程来说，这就可能使文件变得非常大。
- 时间的浪费。如果读进程读数据比写进程写数据快，那么，就可能出现读进程不断地读文件尾部，使读进程做很多无用功。

要克服以上缺点而又使进程间的通信相对简单，管道是一种较好的选择。

所谓管道，是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称 pipe 文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入管道；而接收管道输出的接收进程（即读进程），可从管道中接收数据。由于发送进程和接收进程是利用管道进行通信的，故又称管道通信。这种方式首创于 UNIX 系统，因它能传送大量的数据，且很有效，故很多操作系统都引入了这种通信方式，Linux 也不例外。

为了协调双方的通信，管道通信机制必须提供以下 3 方面的协调能力。

- 互斥。当一个进程正在对 pipe 进行读/写操作时，另一个进程必须等待。

- 同步。当写（输入）进程把一定数量（如 4KB）数据写入 pipe 后，便去睡眠等待，直到读（输出）进程取走数据后，再把它唤醒。当读进程读到一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将它唤醒。
- 对方是否存在。只有确定对方已存在时，才能进行通信。

7.1.1 Linux 管道的实现机制

在 Linux 中，管道是一种使用非常频繁的通信机制。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现如下所述。

- 限制管道的大小。实际上，管道是一个固定大小的缓冲区。在 Linux 中，该缓冲区的大小为 1 页，即 4KB，使得它的大小不像文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的 write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 write()调用写。
- 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的 read()调用将默认地被阻塞，等待某些数据被写入，这解决了 read()调用返回文件结束的问题。

注意，从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

1．管道的结构

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的 file 结构和 VFS 的索引节点 inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。如图 7.1 所示。

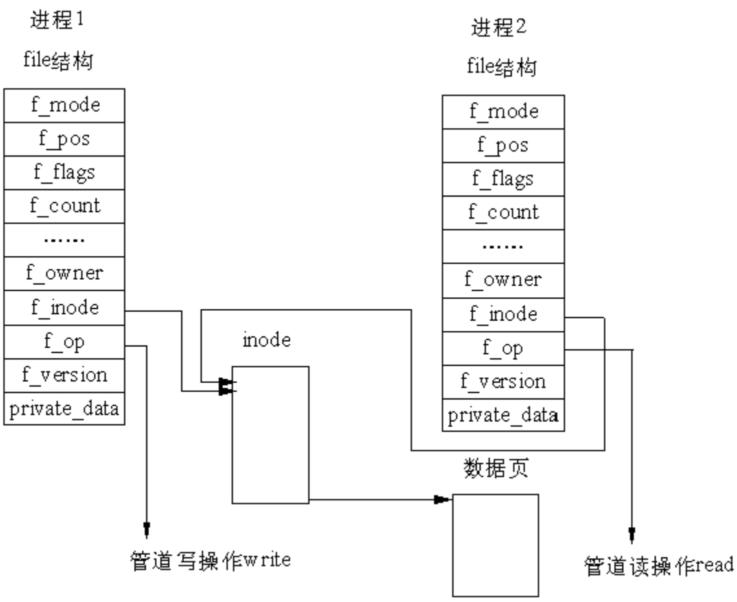


图 7.1 管道结构示意图

图 7.1 中有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，而另一个是从管道中读出数据的例程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

2．管道的读写

管道实现的源代码在 fs/pipe.c 中，在 pipe.c 中有很多函数，其中有两个函数比较重要，即管道读函数 pipe_read()和管道写函数 pipe_wrtie()。管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。当然，内核必须利用一定的机制同步对管道的访问，为此，内核使用了锁、等待队列和信号。

当写进程向管道中写入时，它利用标准的库函数 write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，同时满足如下条件时，才能进行实际的内存复制工作：

- 内存中有足够的空间可容纳所有要写入的数据；
- 内存没有被读程序锁定。

如果同时满足上述条件，写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。

管道的读取过程和写入过程类似。但是，进程可以在没有数据或内存被锁定时立即返回错误信息，而不是阻塞该进程，这依赖于文件或管道的打开模式。反之，进程可以休眠在索引节点的等待队列中等待写入进程写入数据。当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。

因为管道的实现涉及很多文件的操作，因此，当读者学完有关文件系统的内容后来读 pipe.c 中的代码，你会觉得并不难理解。

7.1.2　管道的应用

管道是利用 pipe()系统调用而不是利用 open()系统调用建立的。pipe()调用的原型是：

int pipe（int fd[2]）

我们看到，有两个文件描述符与管道结合在一起，一个文件描述符用于管道的 read()端，一个文件描述符用于管道的 write()端。由于一个函数调用不能返回两个值，pipe()的参数是指向两个元素的整型数组的指针，它将由调用两个所要求的文件描述符填入。

fd[0]元素将含有管道 read()端的文件描述符，而 fd[1]含有管道 write()端的文件描述符。系统可根据 fd[0]和 fd[1]分别找到对应的 file 结构。在第八章我们会描述 pipe()系统

调用的实现机制。

注意，在 pipe 的参数中，没有路径名，这表明，创建管道并不像创建文件一样，要为它创建一个目录连接。这样做的好处是，其他现存的进程无法得到该管道的文件描述符，从而不能访问它。那么，两个进程如何使用一个管道来通信呢？

我们知道，fork()和 exec()系统调用可以保证文件描述符的复制品既可供双亲进程使用,也可供它的子女进程使用。也就是说,一个进程用 pipe()系统调用创建管道,然后用 fork()调用创建一个或多个进程，那么，管道的文件描述符将可供所有这些进程使用。pipe()系统调用的具体实现将在下一章介绍。

这里更明确的含义是：一个普通的管道仅可供具有共同祖先的两个进程之间共享，并且这个祖先必须已经建立了供它们使用的管道。

注意，在管道中的数据始终以和写数据相同的次序来进行读，这表示 lseek()系统调用对管道不起作用。

下面给出在两个进程之间设置和使用管道的简单程序：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
       char    readbuffer[80];

       pipe (fd) ;

        if ( (childpid = fork())  == -1)
        {
                printf ("Error:fork") ;
                exit (1) ;
        }

        if (childpid == 0)         /* 子进程是管道的写进程 */
       {
               close (fd[0]) ;      /*关闭管道的读端  */
               write (fd[1], string, strlen (string) ) ;
               exit (0) ;
        }
        else                            /* 父进程是管道的读进程  */
        {
                   close (fd[1]) ;    /*关闭管道的写端 */
               nbytes = read (fd[0], readbuffer, sizeof (readbuffer) ) ;
                printf ("Received string: %s", readbuffer) ;
        }
        return (0) ;
}
```

注意，在这个例子中，为什么这两个进程都关闭它所不需的管道端呢？这是因为写进程完全关闭管道端时，文件结束的条件被正确地传递给读进程。而读进程完全关闭管道端时，

写进程无需等待继续写数据。

阻塞读和写分别成为对空和满管道的默认操作，这些默认操作也可以改变，这就需要调用 fcntl()系统调用，对管道文件描述符设置 O_NONBLOCK 标志可以忽略默认操作：

```
# include  <fcntl.h>

fcntl (fd,F_SETFL,O_NONBlOCK) ;
```

7.1.3　命名管道 CFIFOC

Linux 还支持另外一种管道形式，称为命名管道，或 FIFO，这是因为这种管道的操作方式基于“先进先出”原理。上面讲述的管道类型也被称为“匿名管道”。命名管道中，首先写入管道的数据是首先被读出的数据。匿名管道是临时对象，而 FIFO 则是文件系统的真正实体，如果进程有足够的权限就可以使用 FIFO。FIFO 和匿名管道的数据结构以及操作极其类似，二者的主要区别在于，FIFO 在使用之前就已经存在，用户可打开或关闭 FIFO；而匿名管道只在操作时存在，因而是临时对象。

为了创建先进先出文件，可以从 shell 提示符使用 mknod 命令或可以在程序中使用 mknod()系统调用。

mknod()系统调用的原型为：

```
#include  <sys/type.h>
#inlcude  <sys/state.h>
#include  <fcntl.h>
#include  <unistd.h>

int mknod (char *pathname,node_t mode, dev_t dev) ;
```

其中 pathname 是被创建的文件名称，mode 表示将在该文件上设置的权限位和将被创建的文件类型（在此情况下为 S_IFIFO），dev 是当创建设备特殊文件时使用的一个值。因此，对于先进先出文件它的值为 0。

一旦先进先出文件已经被创建，它可以由任何具有适当权限的进程利用标准的 open()系统调用加以访问。当用 open()调用打开时，一个先进先出文件和一个匿名管道具有同样的基本功能。即当管道是空的时候，read()调用被阻塞。当管道是满的时候，write()等待被阻塞，并且当用 fcntl()设置 O_NONBLOCK 标志时，将引起 read()调用和 write()调用立即返回。在它们已被阻塞的情况下，带有一个 EAGAIN 错误信息。

由于命名管道可以被很多无关系的进程同时访问，那么，在有多个读进程和/或多个写进程的应用中使用 FIFO 是非常有用的。

多个进程写一个管道会出现这样的问题，即多个进程所写的数据混在一起怎么办？幸好系统有这样的规则：一个 write()调用可以写管道能容纳（Linux 为 4KB）的任意个字节，系统将保证这些数据是分开的。这表示多个写操作的数据在 FIFO 文件中并不混合而将被维持分离的信息。

7.2　信号（signal）

尽管大多数进程间通信是计划好的，但同时还需要处理不可预知的通信问题。例如，用户使用文本编辑器要求列出一个大文件的全部内容，但随即他认识到该操作并不重要，这时就需要一种方法来中止编辑器的工作，例如，用户可以通过 DEL 键作到这点，按 DEL 键实际上是向编辑器发送一个信号，编辑器收到此信号即停止打印文件的内容。信号还可用来报告硬件捕获到的特定的陷入，如非法指令或浮点运算溢出，超时也是通过信号来实现的。

实际上，信号机制是在软件层次上对中断机制的模拟。从概念上讲，一个进程接受到一个信号与一个处理器接受到一个中断请求是一样的。一个进程所接收到的信号可以来自其他进程，可以来自外部事件，也可以来自进程自身。最重要的是，信号和中断都是“异步”的。处理器在执行一段程序时并不需要停下来等待中断的发生，也不知道中断会何时发生。信号也一样，一个进程并不需要通过一个什么样的操作来等待信号的到达，也不知道信号会什么时候到达。

7.2.1　信号种类

每一种信号都给予一个符号名，对 32 位的 i386 平台而言，一个字为 32 位，因此信号有 32 种，而对 64 位的 Alpha AXP 平台而言，每个字为 64 位，因此信号最多可有 64 种。Linux 定义了 i386 的 32 个信号，在 include/asm/signal.h 中定义。表 7.1 给出常用的符号名、描述和它们的信号值。

表 7.1　　信号和其对应的值

符号名	描述	信号值
SIGHUP	在控制终端上发生的结束信号	1
SIGINT	中断，用户键入 CTRL－C 时发送	2
SIGQUIT	从键盘来的中断（ctrl_c）信号	3
SIGILL	非法指令	4
SIGTRAP	跟踪陷入	5
SIGABRT	非正常结束，程序调用 abort 时发送	6
SIGIOT	IOT 指令	6
SIGBUS	总线超时	7
SIGFPE	浮点异常	8
SIGKILL	杀死进程（不能被捕或忽略）	9
SIGUSR1	用户定义信号#1	10
SIGSEGV	段违法	11
SIGUSR2	用户定义信号#2	12
SIGPIPE	向无人读到的管道写	13
SIGALRM	定时器告警，时间到	14

SIGTERM	Kill 发出的软件结束信号	15
SIGCHLD	子程序结束或停止	17
SIGCONT	如果已停止则续继	18
SIGSTOP	停止信号	19

续表

符号名	描述	信号值
SIGTSTP	交互停止信号	20
SIGTTIN	后台进程想读	21
SIGTTOU	后台进程想写	22
SIGPWR	电源失效	30

每种信号类型都有对应的信号处理程序（也叫信号的操作），就好像每个中断都有一个中断服务例程一样。大多数信号的默认操作是结束接收信号的进程。然而，一个进程通常可以请求系统采取某些代替的操作，各种代替操作如下所述。

（1）忽略信号。随着这一选项的设置，进程将忽略信号的出现。有两个信号不可以被忽略：SIGKILL，它将结束进程；SIGSTOP，它是作业控制机制的一部分，将挂起作业的执行。

（2）恢复信号的默认操作。

（3）执行一个预先安排的信号处理函数。进程可以登记特殊的信号处理函数。当进程收到信号时，信号处理函数将像中断服务例程一样被调用，当从该信号处理函数返回时，控制被返回给主程序，并且继续正常执行。

但是，信号和中断有所不同。中断的响应和处理都发生在内核空间，而信号的响应发生在内核空间，信号处理程序的执行却发生在用户空间。

那么，什么时候检测和响应信号呢？通常发生在以下两种情况下：

（1）当前进程由于系统调用、中断或异常而进入内核空间以后，从内核空间返回到用户空间前夕；

（2）当前进程在内核中进入睡眠以后刚被唤醒的时候，由于检测到信号的存在而提前返回到用户空间。

当有信号要响应时，处理器执行路线的示意图如图 7.2 所示。

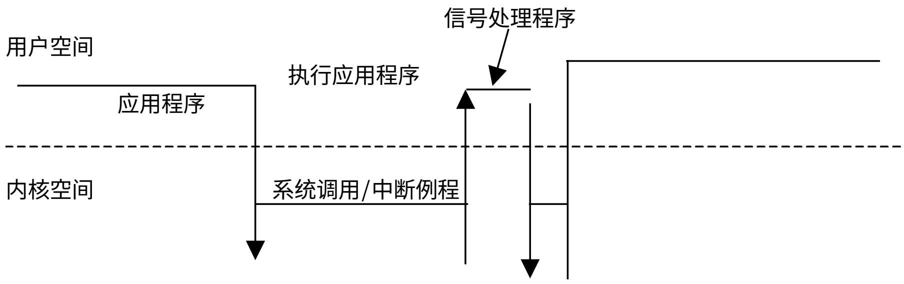


图 7.2　信号的检测及处理流程示意图

从图 7.2 中可以看出，当前进程在用户态执行的过程中，陷入系统调用或中断服务例程，

于是，当前进程从用户态切换到内核态；当处理完系统调用要返回到用户态前夕，发现有信号处理程序需要执行，于是，又从内核态切换到用户态；当执行完信号处理程序后，并不是接着就在用户态执行应用程序，而是还要返回到内核态。为什么还要返回到内核态呢？这是因为此时还没有真正从系统调用返回到用户态，于是从信号处理程序返回到内核态就是为了处理从系统调用到用户态的返回。读者能否想出更好的办法来处理这种状态的来回切换呢。

7.2.2 信号掩码

在 POSIX 下，每个进程有一个信号掩码（Signal Mask）。简单地说，信号掩码是一个“位图”，其中每一位都对应着一种信号。如果位图中的某一位为 1，就表示在执行当前信号的处理程序期间相应的信号暂时被“屏蔽”，使得在执行的过程中不会嵌套地响应那种信号。

为什么对某一信号进行屏蔽呢？我们来看一下对 CTRL+C 的处理。大家知道，当一个程序正在运行时，在键盘上按一下 CTRL+C，内核就会向相应的进程发出一个 SIGINT 信号，而对这个信号的默认操作就是通过 do_exit()结束该进程的运行。但是，有些应用程序可能对 CTRL+C 有自己的处理，所以就要为 SIGINT 另行设置一个处理程序，使它指向应用程序中的一个函数，在那个函数中对 CTRL+C 这个事件作出响应。但是，在实践中却发现，两次 CTRL+C 事件往往过于密集，有时候刚刚进入第 1 个信号的处理程序，第 2 个 SIGINT 信号就到达了，而第 2 个信号的默认操作是杀死进程，这样，第 1 个信号的处理程序根本没有执行完。为了避免这种情况的出现，就在执行一个信号处理程序的过程中将该种信号自动屏蔽掉。所谓“屏蔽”，与将信号忽略是不同的，它只是将信号暂时“遮盖”一下，一旦屏蔽去掉，已到达的信号又继续得到处理。

Linux 内核中有一个专门的函数集合来执行设置和修改信号掩码，它们放在 kernel/signal.c 中，其函数形式和功能如下：

函数形式	功能
int sigemptyset（sigset_t *mask）	清所有信号掩码的阻塞标志
int sigfillset（sigset_t *mask，int signum）	设置所有信号掩码的阻塞标志
int sigdelset（sigset_t *mask，int signum）	删除个别信号阻塞
int sigaddset（sigset_t *mask，int signum）	增加个别信号阻塞
int sigisnumber（sigset_t *mask，int signum）	确定特定的信号是否在掩码中被标志为阻塞

另外，进程也可以利用 sigprocmask()系统调用改变和检查自己的信号掩码的值，其实现代码在 kernel/signal.c 中，原型为：

```
int sys_sigprocmask（int how，sigset_t *set，sigset_t *oset）
```

其中，set 是指向信号掩码的指针，进程的信号掩码是根据参数 how 的取值设置成 set。参数 how 的取值及含义如下：

SIG_BOLCK	set 规定附加的阻塞信号
SIG_UNBOCK	set 规定一组不予阻塞的信号
SIG_SETBLOCK	set 变成新进程的信号掩码

用一段代码来说明这个问题：

```
switch (how) {
case SIG_BLOCK:
       current->blocked |= new_set;
           break;
 case SIG_UNBLOCK:
         current->blocked &= ~new_set;
         break;
 case SIG_SETMASK:
         current->blocked = new_set;
         break;
 default:
     return -EINVAL;
    }
```

其中 current 为指向当前进程 task_struc 结构的指针。

第 3 个参数 oset 也是指向信号掩码的指针，它将包含以前的信号掩码值，使得在必要的时候，可以恢复它。

进程可以用 sigpending()系统调用来检查是否有挂起的阻塞信号。

7.2.3　系统调用

除了 signal()系统调用，Linux 还提供关于信号的系统调用如下：

调用原型	功能
int sigaction (sig,&handler,&oldhandler)	定义对信号的处理操作
int sigreturn (&context)	从信号返回
int sigprocmask (int how, sigset_t *mask, sigset_t *old)	检查或修改信号屏蔽
int sigpending (sigset_t mask)	替换信号掩码并使进程挂起
int kill (pid_t pid, int sig)	发送信号到进程
long alarm (long secs)	设置事件闹钟
int pause (void)	将调用进程挂起直到下一个进程

其中 sigset_t 定义为：

```
typedef unsigned long sigset_t;       /* 至少 32 位*/
```

下面介绍几个典型的系统调用。

1. ill 系统调用

从前面的叙述可以看到，一个进程接收到的信号，或者是由异常的错误产生（如浮点异常），或者是用户在键盘上用中断和退出信号干涉而产生，那么，一个进程能否给另一个进程发送信号？回答是肯定的，但发送者进程必须有适当的权限。Kill()系统调用可以完成此任务：

```
int kill (pid_t pid, int sig)
```

参数 sig 规定发送哪一个信号，参数 pid（进程标识号）规定把信号发送到何处，pid

各种不同值具有下列意义：

pid>0　信号 sig 发送给进程标识号为 pid 的进程；

pid=0　设调用 kill()的进程其组标识号为 p，则把信号 sig 发送给与 p 相等的其他所有进程；

pid=-1　inux 规定把信号 sig 发送给系统中除去 init 进程和调用者以外的所有进程；

pid<-1　信号发送给进程组-pid 中的所有进程。

为了用 kill()发送信号，调用进程的有效用户 ID 必须是 root，或者必须和接收进程的实际或有效用户 ID 相同。

2．ause()和 alarm()系统调用

当一个进程需要等待另一个进程完成某项操作时，它将执行 pause()调用，当这项操作已完成时，另一个进程可以发送一个预约的信号给这一暂停的进程，它将强迫 pause()返回，并且允许收到信号的进程恢复执行，知道它正在等待的事件现在已经出现。

对于许多实际应用，需要在一段指定时间后，中断进程的原有操作，以进行某种其他的处理，例如在不可靠的通信线路上重传一个丢失的包，为了处理此类情况，系统提供了 alarm()系统调用。每个进程都有一个闹钟计时器与之相联，在经过预先设置的时间后，进程可以用它来给自己发送 SIGALARM 信号。Alarm()调用只取一个参数 secs，它是在闹钟关闭之前所经过的秒数。如果传递一个 0 值给 alarm()，这将关闭任何当前正在运行的闹钟计时器。

Alarm()返回值是以前的闹钟计时器值，如果当前没有设置任何闹钟计时器，这将是零，或者是当作出该调用时，闹钟的剩余时间。

某些情况下，进程在信号到达之前不要做任何操作，例如一个测验阅读和理解速度的 CAI 系统，它先在屏幕上显示一些课文，然后调用 alarm()设定在 30s 后向自己发送一个信号，以激活程序进行一些处理。当学生阅读课文时，程序无需执行任何操作，它可以采用的一种方法是执行空操作循环以等待时间到，但假如此时系统中还有其他进程运行，这将浪费 CPU 的时间，好的方法是使用 pause()系统调用，它将挂起调用进程直到信号到来，这段时间里别的进程可以使用 CPU。

7.2.4　典型系统调用的实现

sigaction()系统调用的实现较具代表性，它的主要功能为设置信号处理程序，其原型为：

```
int sys_sigaction (int signum, const struct sigaction * action,
                   struct sigaction * oldaction)
```

其中，sigaction 数据结构在 include/asm/signal.h 中定义，其格式为：

```
struct sigaction {
        __sighandler_t  sa_handler;
    sigset_t       sa_mask;
          unsigned long  sa_flags;
         void  (*sa_restorer) (void) ;
            };
```

其中__sighandler_t 定义为：

```
typedef void (*__sighandler_t) (int);
```

在这个结构中， sa_handler 为指向处理函数的指针，sa_mask 是信号掩码，当该信号 signum 出现时，这个掩码就被逻辑或到接收进程的信号掩码中。当信号处理程序执行时，这个掩码保持有效。Sa_flags 域是几个位标志的逻辑或（OR）组合，其中两个主要的标志是：

SA_ONESHOT　信号出现时，将信号操作置为默认操作；

SA_NOMASK　忽略 sigaction 结构的 sa_mask 域。

Linux 中定义的信号处理的 3 种类型为：

```
#define SIG_DFL  ((__sighandler_t) 0)      * 缺省的信号处理*/
#define SIG_IGN  ((__sighandler_t) 1)      *忽略这个信号 */
#define SIG_ERR  ((__sighandler_t) -1) *从信号返回错误 */
```

下面是 sigaction()系统调用在内核中实现的代码及解释。

```
int sys_sigaction (int signum, const struct sigaction * action,
                   struct sigaction * oldaction)
{
    struct sigaction new_sa, *p;

      if (signum<1 || signum>32)
           return -EINVAL;
      /* 信号的值不在 1～32 之间，则出错 */
      if (signum==SIGKILL || signum==SIGSTOP)
           return -EINVAL;
      /*SIGKILL 和 SIGSTOP 不能设置信号处理程序 */
       p = signum - 1 + current->sig->action;
          /*在当前进程中，指向信号 signum 的 action 的指针 */
       if (action) {
      int err = verify_area (VERIFY_READ, action, sizeof (*action));
         /* 验证给 action 在用户空间分配的地址的有效性 */
          if (err)
               return err;
      memcpy_fromfs (&new_sa, action, sizeof (struct sigaction));
        /* 把 actoin 的内容从用户空间拷贝到内核空间*/
       new_sa.sa_mask |= _S (signum);
        /* 把信号 signum 加到掩码中 */
       if (new_sa.sa_flags & SA_NOMASK)
           new_sa.sa_mask &= ~_S (signum);
/* 如果标志为 SA_NOMASK，当信号 signum 出现时，将它的操作置为默认操作 */
       new_sa.sa_mask &= _BLOCKABLE;
      /* 不能阻塞 SIGKILL 和 SIGSTOP  */
if (new_sa.sa_handler != SIG_DFL && new_sa.sa_handler !=SIG_IGN) {
           err = verify_area (VERIFY_READ, new_sa.sa_handler, 1);
/* 当处理程序不是信号默认的处理操作，并且 signum 信号不能被忽略时，验证给信号处理程序分配
空间的有效性 */
       if (err)
            return err;        }
       }
   if (oldaction) {
```

```
    int err = verify_area (VERIFY_WRITE, oldaction,   sizeof (*oldaction) ) ;
        if  (err)
                return err;
        memcpy_tofs (oldaction, p, sizeof (struct sigaction) ) ;
         /* 恢复原来的信号处理程序 */
                    }
        if  (action)  {
           *p = new_sa;
           check_pending (signum) ;
                 }
          return 0;
  }
```

Linux 可以将各种信号发送给程序，以表示程序故障、用户请求的中断、其他各种情况等。通过对 sigaction()系统调用源代码的分析，有助于灵活应用信号的系统调用。

7.2.5 进程与信号的关系

Linux 内核中不存在任何机制用来区分不同信号的优先级。也就是说，当同时有多个信号发出时，进程可能会以任意顺序接收到信号并进行处理。另外，如果进程在处理某个信号之前，又有相同的信号发出，则进程只能接收到一个信号。产生上述现象的原因与内核对信号的实现有关。

系统在 task_struct 结构中利用两个域分别记录当前挂起的信号（Signal）以及当前阻塞的信号（Blocked）。挂起的信号指尚未进行处理的信号。阻塞的信号指进程当前不处理的信号，如果产生了某个当前被阻塞的信号，则该信号会一直保持挂起，直到该信号不再被阻塞为止。除了 SIGKILL 和 SIGSTOP 信号外，所有的信号均可以被阻塞，信号的阻塞可通过系统调用 sigprocmask()实现。每个进程的 task_struct 结构中还包含了一个指向 sigaction 结构数组的指针，该结构数组中的信息实际指定了进程处理所有信号的方式。如果某个 sigaction 结构中包含有处理信号的例程地址，则由该处理例程处理该信号；反之，则根据结构中的一个标志或者由内核进行默认处理，或者只是忽略该信号。通过系统调用 sigaction()，进程可以修改 sigaction 结构数组的信息，从而指定进程处理信号的方式。

进程不能向系统中所有的进程发送信号，一般而言，除系统和超级用户外，普通进程只能向具有相同 uid 和 gid 的进程，或者处于同一进程组的进程发送信号。当有信号产生时，内核将进程 task_struct 的 signal 字中的相应位设置为 1。系统不对置位之前该位已经为 1 的情况进行处理，因而进程无法接收到前一次信号。如果进程当前没有阻塞该信号，并且进程正处于可中断的等待状态（INTERRUPTIBLE），则内核将该进程的状态改变为运行(RUNNING)，并放置在运行队列中。这样，调度程序在进行调度时，就有可能选择该进程运行，从而可以让进程处理该信号。

发送给某个进程的信号并不会立即得到处理，相反，只有该进程再次运行时，才有机会处理该信号。每次进程从系统调用中退出时，内核会检查它的 signal 和 block 字段，如果有任何一个未被阻塞的信号发出，内核就根据 sigaction 结构数组中的信息进行处理。处理过程如下。

（1）检查对应的 sigaction 结构，如果该信号不是 SIGKILL 或 SIGSTOP 信号，且被忽略，则不处理该信号。

（2）如果该信号利用默认的处理程序处理，则由内核处理该信号，否则转向第（3）步。

（3）该信号由进程自己的处理程序处理，内核将修改当前进程的调用堆栈，并将进程的程序计数寄存器修改为信号处理程序的入口地址。此后，指令将跳转到信号处理程序，当从信号处理程序中返回时，实际就返回了进程的用户模式部分。

Linux 是 与 POSIX 兼容的，因此，进程在处理某个信号时，还可以修改进程的 blocked 掩码。但是，当信号处理程序返回时，blocked 值必须恢复为原有的掩码值，这一任务由内核的 sigaction()函数完成。Linux 在进程的调用堆栈帧中添加了对清理程序的调用，该清理程序可以恢复原有的 blocked 掩码值。当内核在处理信号时，可能同时有多个信号需要由用户处理程序处理，这时，Linux 内核可以将所有的信号处理程序地址推入堆栈中，而当所有的信号处理完毕后，调用清理程序恢复原先的 blocked 值。

7.2.6　信号举例

下面通过 Linux 提供的系统调用 signal()，来说明如何执行一个预先安排好的信号处理函数。Signal()调用的原型是：

```
#include <signal.h>
#include <unistd.h>

void （* signal（int signum, void（*handler）（int））） （int）;
```

signal()的返回值是指向一个函数的指针，该函数的参数为一个整数，无返回值，下面是用户级程序的一段代码。

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int ctrl_c_count=0;
void （* old_handler） （INT）;
void ctrl_c（int）;

main()
{
   int c;

   old_handler = signal（SIGINT,ctrl_c）;

      while （（c=getchar()）! = '\n'）;

      printf（"ctrl-c count = %d\n",ctrl_c_count）;

       （void） signal（SIGINT,old_handler）;
     }

   void ctrl_c（int signum）
```

```
    {
      (void) signal (SIGINT,ctrl_c)
     ++ctrl_c;
   }
```

程序说明：这个程序是从键盘获得字符，直到换行符为止，然后进入无限循环。这里，程序安排了捕获 ctrl_c 信号（SIGINT），并且利用 SIGINT 来执行一个 ctrl_c 的处理函数。当在键盘上敲入一个换行符时，SIGINT 原来的操作（很可能是默认操作）才被恢复。Main()函数中的第一个语句完成设置信号处理程序：

```
old_handler = signal (SIGINT,ctrl_c) ;
```

signal()的两个参数是：信号值，这里是键盘中断信号 SIGINT，以及一个指向函数的指针，这里是 ctrl_c，当这个中断信号出现时，将调用该函数。Signal()调用返回旧的信号处理程序的地址，在此它被赋给变量 older_handler，使得原来的信号处理程序稍后可以被恢复。

一旦信号处理程序放在应放的位置，进程收到任何中断（SIGINT）信号将引起信号处理函数的执行。这个函数增加 ctrl_c_count 变量的值以保持对 SIGINT 事件出现次数的计数。注意信号处理函数也执行另一个 signal()调用，它重新建立 SIGINT 信号和 ctrl_c 函数之间的联系。这是必需的，因为当信号出现时，用 signal()调用设置的信号处理程序被自动恢复为默认操作，使得随后的同一信号将只执行信号的默认操作。

7.3 System V 的 IPC 机制

为了提供与其他系统的兼容性，Linux 也支持 3 种 system V 的进程间通信机制：消息、信号量（semaphores）和共享内存，Linux 对这些机制的实施大同小异。我们把信号量、消息和共享内存统称 System V IPC 的对象，每一个对象都具有同样类型的接口，即系统调用。就像每个文件都有一个打开文件号一样，每个对象也都有唯一的识别号，进程可以通过系统调用传递的识别号来存取这些对象，与文件的存取一样，对这些对象的存取也要验证存取权限，System V IPC 可以通过系统调用对对象的创建者设置这些对象的存取权限。

在 Linux 内核中，System V IPC 的所有对象有一个公共的数据结构 pc_perm 结构，它是 IPC 对象的权限描述，在 linux/ipc.h 中定义如下：

```
struct ipc_perm
{
   key_t  key;      /* 键 */
   ushort uid;      /* 对象拥有者对应进程的有效用户识别号和有效组识别号 */
   ushort gid;
  ushort cuid;      /* 对象创建者对应进程的有效用户识别号和有效组识别号 */
   ushort cgid;
   ushort mode;          /*  存取模式 */
   ushort seq;      /* 序列号 */
};
```

在这个结构中，要进一步说明的是键（key）。键和识别号指的是不同的东西。系统支持两种键：公有和私有。如果键是公有的，则系统中所有的进程通过权限检查后，均可以找到 System V IPC 对象的识别号。如果键是公有的，则键值为 0，说明每个进程都可以用键值

0 建立一个专供其私用的对象。注意，对 System V IPC 对象的引用是通过识别号而不是通过键，从后面的系统调用中可了解这一点。

7.3.1　信号量

信号量及信号量上的操作是 E.W.Dijkstra 在 1965 年提出的一种解决同步、互斥问题的较通用的方法，并在很多操作系统中得以实现，Linux 改进并实现了这种机制。

信号量（semaphore）实际是一个整数，它的值由多个进程进行测试（test）和设置（set）。就每个进程所关心的测试和设置操作而言，这两个操作是不可中断的，或称“原子”操作，即一旦开始直到两个操作全部完成。测试和设置操作的结果是：信号量的当前值和设置值相加，其和或者是正或者为负。根据测试和设置操作的结果，一个进程可能必须睡眠，直到有另一个进程改变信号量的值。

信号量可用来实现所谓的“临界区”的互斥使用，临界区指同一时刻只能有一个进程执行其中代码的代码段。为了进一步理解信号量的使用，下面我们举例说明。

假设你有很多相互协作的进程，它们正在读或写一个数据文件中的记录。你可能希望严格协调对这个文件的存取，于是你使用初始值为 1 的信号量，在这个信号量上实施两个操作，首先测试并且给信号量的值减 1，然后测试并给信号量的值加 1。当第 1 个进程存取文件时，它把信号量的值减 1，并获得成功，信号量的值现在变为 0，这个进程可以继续执行并存取数据文件。但是，如果另外一个进程也希望存取这个文件，那么它也把信号量的值减 1，结果是不能存取这个文件，因为信号量的值变为-1。这个进程将被挂起，直到第一个进程完成对数据文件的存取。当第 1 个进程完成对数据文件的存取，它将增加信号量的值，使它重新变为 1，现在，等待的进程被唤醒，它对信号量的减 1 操作将获得成功。

上述的进程互斥问题，是针对进程之间要共享一个临界资源而言的，信号量的初值为 1。实际上，信号量作为资源计数器，它的初值可以是任何正整数，其初值不一定为 0 或 1。另外，如果一个进程要先获得两个或多个的共享资源后才能执行的话，那么，相应地也需要多个信号量，而多个进程要分别获得多个临界资源后方能运行，这就是信号量集合机制，Linux 讨论的就是信号量集合问题。

1．信号量的数据结构

Linux 中信号量是通过内核提供的一系列数据结构实现的，这些数据结构存在于内核空间，对它们的分析是充分理解信号量及利用信号量实现进程间通信的基础，下面先给出信号量的数据结构（存在于 include/linux/sem.h 中），其他一些数据结构将在相关的系统调用中介绍。

（1）系统中每个信号量的数据结构（sem）

```
struct sem {
  int   semval;          /* 信号量的当前值   */
            int    sempid;          /*在信号量上最后一次操作的进程识别号 *
         };
```

（2）系统中表示信号量集合（set）的数据结构（semid_ds）

```
struct semid_ds {
```

```
      struct ipc_perm sem_perm;         /*  IPC 权限 */
      long        sem_otime;                /* 最后一次对信号量操作（semop）的时间 */
    long        sem_ctime;              /* 对这个结构最后一次修改的时间 */
      struct sem      *sem_base;        /* 在信号量数组中指向第一个信号量的指针 */
      struct sem_queue *sem_pending;              /* 待处理的挂起操作*/
      struct sem_queue **sem_pending_last;  /* 最后一个挂起操作 */
      struct sem_undo *undo;                      /* 在这个数组上的 undo 请求 */
    ushort          sem_nsems;                          /* 在信号量数组上的信号量号 */
      };
```

(3) 系统中每一信号量集合的队列结构（sem_queue）

```
struct sem_queue {
    struct sem_queue *  next;     /* 队列中下一个节点 */
    struct sem_queue ** prev;      /* 队列中前一个节点，*（q->prev） == q */
    struct wait_queue * sleeper;     /* 正在睡眠的进程 */
    struct sem_undo *   undo;      /* undo 结构*/
    int             pid;  /* 请求进程的进程识别号 */
    int             status;    /* 操作的完成状态 */
    struct semid_ds *   sma;  /*有操作的信号量集合数组 */
    struct sembuf *     sops;      /* 挂起操作的数组 */
    int        nsops;     /* 操作的个数 */
};
```

(4) 几个主要数据结构之间的关系

从图 7.3 可以看出，semid_ds 结构的 sem_base 指向一个信号量数组，允许操作这些信号量集合的进程可以利用系统调用执行操作 。注意，信号量与信号量集合的区别，从上面可以看出，信号量用“ sem ” 结构描述，而信号量集合用“ semid_ds ”结构描述，实际上，在后面的讨论中，我们以信号量集合为讨论的主要对象。下面我们给出这几个结构之间的关系，如图 7.3 所示。

Linux 对信号量的这种实现机制，是为了与消息和共享内存的实现机制保持一致，但信号量是这 3 者中最难理解的，因此我们将结合系统调用做进一步的介绍，通过对系统调用的深入分析，我们可以较清楚地了解内核对信号量的实现机制。

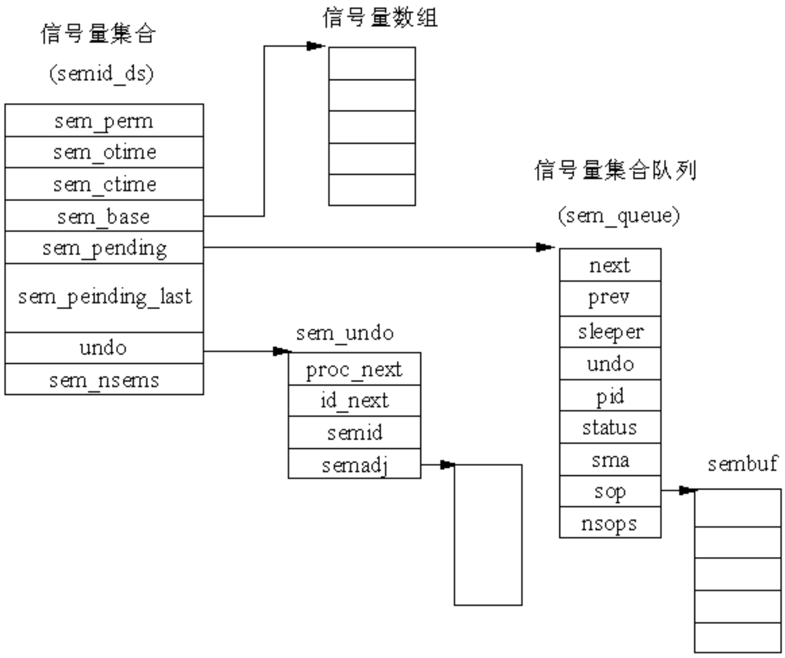


图 7.3　System V IPC 信号量数据结构之间的关系

2．系统调用：semget()

为了创建一个新的信号量集合，或者存取一个已存在的集合，要使用 segget()系统调用，其描述如下：

原型：int semget （ key_t key, int nsems, int semflg ）；

返回值：如果成功，则返回信号量集合的 IPC 识别号；

如果为-1，则出现错误。

semget()中的第 1 个参数是键值，这个键值要与已有的键值进行比较，已有的键值指在内核中已存在的其他信号量集合的键值。对信号量集合的打开或存取操作依赖于 semflg 参数的取值。

- IPC_CREAT ：如果内核中没有新创建的信号量集合，则创建它。
- IPC_EXCL ：当与 IPC_CREAT 一起使用时，如果信号量集合已经存在，则创建失败。

如果 IPC_CREAT 单独使用，semget()为一个新创建的集合返回标识号，或者返回具有相同键值的已存在集合的标识号。如果 IPC_EXCL 与 IPC_CREAT 一起使用，要么创建一个新的集合，要么对已存在的集合返回-1。IPC_EXCL 单独是没有用的，当与 IPC_CREAT 结合起来使用时，可以保证新创建集合的打开和存取。

作为 System V IPC 的其他形式，一种可选项是把一个八进制与掩码或，形成信号量集合的存取权限。

第 3 个参数 nsems 指的是在新创建的集合中信号量的个数。其最大值在“ linux/sem.h ”中定义：

```
#define SEMMSL  250             /* <= 8 000 max num of semaphores per id */
```

注意，如果你是显式地打开一个现有的集合，则 nsems 参数可以忽略。

下面举例说明。

```
int open_semaphore_set ( key_t keyval, int numsems )
{
        int     sid;

        if  ( ! numsems )
                return (-1) ;

        if ( (sid = semget ( keyval, numsems, IPC_CREAT | 0660 ) )  == -1)
        {
                return (-1) ;
        }

        return (sid) ;
}
```

注意，这个例子显式地用了 0660 权限。这个函数要么返回一个集合的标识号，要么返回-1 而出错。键值必须传递给它，信号量的个数也传递给它，这是因为如果创建成功则要分配空间。

3．系统调用：semop()

原型： int semop (int semid, struct sembuf *sops, unsigned nsops) ;

返回：如果所有的操作都执行，则成功返回 0。

如果为-1，则出错。

semop()中的第 1 个参数（semid）是集合的识别号（可以由 semget()系统调用得到）。第 2 个参数（sops）是一个指针，它指向在集合上执行操作的数组。而第 3 个参数（nsops）是在那个数组上操作的个数。

sops 参数指向类型为 sembuf 的一个数组，这个结构在/inclide/linux/sem.h 中声明，是内核中的一个数据结构，描述如下：

```
struct sembuf {
  ushort  sem_num;        /* 在数组中信号量的索引值 */
  short   sem_op;         /* 信号量操作值（正数、负数或 0） */
  short   sem_flg;        /* 操作标志，为 IPC_NOWAIT 或 SEM_UNDO*/
};
```

如果 sem_op 为负数，那么就从信号量的值中减去 sem_op 的绝对值，这意味着进程要获取资源，这些资源是由信号量控制或监控来存取的。如果没有指定 IPC_NOWAIT，那么调用进程睡眠到请求的资源数得到满足（其他的进程可能释放一些资源）。

如果 sem_op 是正数，把它的值加到信号量，这意味着把资源归还给应用程序的集合。

最后，如果 sem_op 为 0，那么调用进程将睡眠到信号量的值也为 0，这相当于一个信号量到达了 100%的利用。

综上所述，Linux 按如下的规则判断是否所有的操作都可以成功：操作值和信号量的当前值相加大于 0，或操作值和当前值均为 0，则操作成功。如果系统调用中指定的所有操作中有一个操作不能成功时，则 Linux 会挂起这一进程。但是，如果操作标志指定这种

情况下不能挂起进程的话，系统调用返回并指明信号量上的操作没有成功，而进程可以继续执行。如果进程被挂起，Linux 必须保存信号量的操作状态并将当前进程放入等待队列。为此，Linux 内核在堆栈中建立一个 sem_queue 结构并填充该结构。新的 sem_queue 结构添加到集合的等待队列中（利用 sem_pending 和 sem_pending_last 指针）。当前进程放入 sem_queue 结构的等待队列中（sleeper）后调用调度程序选择其他的进程运行。

为了进一步解释 semop()调用，让我们来看一个例子。假设我们有一台打印机，一次只能打印一个作业。我们创建一个只有一个信号量的集合（仅一个打印机），并且给信号量的初值为 1（因为一次只能有一个作业）。

每当我们希望把一个作业发送给打印机时，首先要确定这个资源是可用的，可以通过从信号量中获得一个单位而达到此目的。让我们装载一个 sembuf 数组来执行这个操作：

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

从这个初始化结构可以看出，0 表示集合中信号量数组的索引，即在集合中只有一个信号量，-1 表示信号量操作（sem_op），操作标志为 IPC_NOWAIT，表示或者调用进程不用等待可立即执行，或者失败（另一个进程正在打印）。下面是用初始化的 sembuf 结构进行 semop()系统调用的例子：

```
if ( (semop (sid, &sem_lock, 1)  == -1)
        fprintf (stderr,"semop\n") ;
```

第 3 个参数（nsops）是说我们仅仅执行了一个操作（在我们的操作数组中只有一个 sembuf 结构），sid 参数是我们集合的 IPC 识别号。

当我们使用完打印机，我们必须把资源返回给集合，以便其他的进程使用。

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

上面这个初始化结构表示，把 1 加到集合数组的第 0 个元素，换句话说，一个单位资源返回给集合。

4．系统调用 ：semctl()

原型：int semctl （ int semid, int semnum, int cmd, union semun arg ）;

返回值： 成功返回正数，出错返回-1。

注意，semctl()是在集合上执行控制操作。

semctl()的第 1 个参数（semid）是集合的标识号，第 2 个参数（semnum）是将要操作的信号量个数，从本质上说，它是集合的一个索引，对于集合上的第一个信号量，则该值为 0。

- cmd 参数表示在集合上执行的命令，这些命令及解释如表 7.2 所示。
- arg 参数的类型为 semun，这个特殊的联合体在 include/linux/sem.h 中声明，对它的描述如下：

```
/* arg for semctl system calls. */
union semun {
        int val;                /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
        ushort *array;          /* array for GETALL & SETALL */
        struct seminfo *__buf;  /* buffer for IPC_INFO */
        void *__pad;
};
```

这个联合体中，有 3 个成员已经在表 7.1 中提到，剩下的两个成员_buf 和_pad 用在内核中信号量的实现代码，开发者很少用到。事实上，这两个成员是 Linux 操作系统所特有的，在 UINX 中没有。

表 7.2　　cmd 命令及解释

命令	解释
IPC_STAT	从信号量集合上检索 semid_ds 结构，并存到 semun 联合体参数的成员 buf 的地址中
IPC_SET	设置一个信号量集合的 semid_ds 结构中 ipc_perm 域的值，并从 semun 的 buf 中取出值
IPC_RMID	从内核中删除信号量集合
GETALL	从信号量集合中获得所有信号量的值，并把其整数值存到 semun 联合体成员的一个指针数组中
GETNCNT	返回当前等待资源的进程个数
GETPID	返回最后一个执行系统调用 semop()进程的 PID
GETVAL	返回信号量集合内单个信号量的值
GETZCNT	返回当前等待 100%资源利用的进程个数
SETALL	与 GETALL 正好相反
SETVAL	用联合体中 val 成员的值设置信号量集合中单个信号量的值

这个系统调用比较复杂，我们举例说明。

下面这个程序段返回集合上索引为 semnum 对应信号量的值。当用 GETVAL 命令时，最后的参数（semnum）被忽略。

```
int get_sem_val ( int sid, int semnum )
{
        return ( semctl (sid, semnum, GETVAL, 0) ) ;
}
```

关于信号量的 3 个系统调用，我们进行了详细的介绍。从中可以看出，这几个系统调用的实现和使用都和系统内核密切相关，因此，如果在了解内核的基础上，再理解系统调用，相对要简单得多，也深入地多。

5．死锁

和信号量操作相关的概念还有“死锁”。当某个进程修改了信号量而进入临界区之后，却因为崩溃或被“杀死（kill）”而没有退出临界区，这时，其他被挂起在信号量上的进程永远得不到运行机会,这就是所谓的死锁。Linux 通过维护一个信号量数组的调整列表(semadj)来避免这一问题。其基本思想是，当应用这些“调整”时，让信号量的状态退回到操作实施前的状态。

关于调整的描述是在 sem_undo 数据结构中，在 include/linux/sem.h 描述如下：

```
/*每一个任务都有一系列的恢复（undo）请求，当进程退出时，自动执行 undo 请求*/

struct sem_undo {
    struct sem_undo *  proc_next;  /*在这个进程上的下一个 sem_undo 节点 */
    struct sem_undo *  id_next;    /* 在这个信号量集和上的下一个 sem_undo 节点*/
    int    semid;                  /* 信号量集的标识号*/
```

```
    short *          semadj;  /* 信号量数组的调整，每个进程一个*/
};
```

sem_undo 结构也出现在 task_struct 数据结构中。

每一个单独的信号量操作也许要请求得到一次“调整”，Linux 将为每一个信号量数组的每一个进程维护至少一个 sem_undo 结构。如果请求的进程没有这个结构，当必要时则创建它，新创建的 sem_undo 数据结构既在这个进程的 task_struct 数据结构中排队，也在信号量数组的 semid_ds 结构中排队。当对信号量数组上的一个信号量施加操作时，这个操作值的负数与这个信号量的“调整”相加，因此，如果操作值为 2，则把-2 加到这个信号量的“调整”域。

当进程被删除时，Linux 完成了对 sem_undo 数据结构的设置及对信号量数组的调整。如果一个信号量集合被删除，sem_undo 结构依然留在这个进程的 task_struct 结构中，但信号量集合的识别号变为无效。

7.3.2　消息队列

一个或多个进程可向消息队列写入消息，而一个或多个进程可从消息队列中读取消息，这种进程间通信机制通常使用在客户/服务器模型中，客户向服务器发送请求消息，服务器读取消息并执行相应请求。在许多微内核结构的操作系统中，内核和各组件之间的基本通信方式就是消息队列。例如，在 Minlx 操作系统中，内核、I/O 任务、服务器进程和用户进程之间就是通过消息队列实现通信的。

Linux 中的消息可以被描述成在内核地址空间的一个内部链表，每一个消息队列由一个 IPC 的标识号唯一地标识。Linux 为系统中所有的消息队列维护一个 msgque 链表，该链表中的每个指针指向一个 msgid_ds 结构，该结构完整描述一个消息队列。

1．数据结构

(1) 消息缓冲区（msgbuf）

我们在这里要介绍的第一个数据结构是 msgbuf 结构，可以把这个特殊的数据结构看成一个存放消息数据的模板，它在 include/linux/msg.h 中声明，描述如下：

```
/* msgsnd 和 msgrcv 系统调用使用的消息缓冲区*/
struct msgbuf {
    long mtype;         /* 消息的类型，必须为正数 */
    char mtext[1];      /* 消息正文 */
};
```

注意，对于消息数据元素（mtext），不要受其描述的限制。实际上，这个域（mtext）不仅能保存字符数组，而且能保存任何形式的任何数据。这个域本身是任意的，因为这个结构本身可以由应用程序员重新定义：

```
struct my_msgbuf {
        long    mtype;          /* 消息类型 */
        long    request_id;     /* 请求识别号 */
        struct  client info;    /* 客户消息结构 */
};
```

我们看到，消息的类型还是和前面一样，但是结构的剩余部分由两个其他的元素代替，

而且有一个是结构。这就是消息队列的优美之处，内核根本不管传送的是什么样的数据，任何信息都可以传送。

但是，消息的长度还是有限制的，在 Linux 中，给定消息的最大长度在 include/linux/msg.h 中定义如下：

```
#define MSGMAX  8192    /* max size of message  (bytes)  */
```

消息总的长度不能超过 8192 字节，包括 mtype 域，它是 4 字节长。

(2) 消息结构（msg）

内核把每一条消息存储在以 msg 结构为框架的队列中，它在 include/ linux/msg.h 中定义如下：

```
struct msg {
    struct msg *msg_next;   /* 队列上的下一条消息 */
    long  msg_type;          /*消息类型*/
    char *msg_spot;         /* 消息正文的地址 */
    short msg_ts;           /* 消息正文的大小 */
};
```

注意，msg_next 是指向下一条消息的指针，它们在内核地址空间形成一个单链表。

(3) 消息队列结构（msgid_ds）

当在系统中创建每一个消息队列时，内核创建、存储及维护这个结构的一个实例。

```
/* 在系统中的每一个消息队列对应一个 msqid_ds 结构 */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;       /* 队列上第一条消息，即链表头*/
    struct msg *msg_last;        /* 队列中的最后一条消息，即链表尾 */
    time_t msg_stime;                 /* 发送给队列的最后一条消息的时间 */
    time_t msg_rtime;            /* 从消息队列接收到的最后一条消息的时间 */
    time_t msg_ctime;                 /* 最后修改队列的时间*/
    ushort msg_cbytes;           /*队列上所有消息总的字节数 */
    ushort msg_qnum;             /*在当前队列上消息的个数 */
    ushort msg_qbytes;           /* 队列最大的字节数 */
    ushort msg_lspid;            /* 发送最后一条消息的进程的 pid */
    ushort msg_lrpid;            /* 接收最后一条消息的进程的 pid */
};
```

2．系统调用: msgget()

为了创建一个新的消息队列，或存取一个已经存在的队列，要使用 msgget()系统调用。

原型: int msgget （ key_t key, int msgflg ）;

返回: 成功，则返回消息队列识别号，失败，则返回-1。

semget()中的第一个参数是键值，这个键值要与现有的键值进行比较，现有的键值指在内核中已存在的其他消息队列的键值。对消息队列的打开或存取操作依赖于 msgflg 参数的取值。

- IPC_CREAT ：如果这个队列在内核中不存在，则创建它。
- IPC_EXCL ：当与 IPC_CREAT 一起使用时，如果这个队列已存在，则创建失败。

如果 IPC_CREAT 单独使用，semget()为一个新创建的消息队列返回标识号，或者返回具有相同键值的已存在队列的标识号。如果 IPC_EXCL 与 IPC_CREAT 一起使用，要么创建一个新

的队列，要么对已存在的队列返回-1。IPC_EXCL 不能单独使用，当与 IPC_CREAT 结合起来使用时，可以保证新创建队列的打开和存取。

与文件系统的存取权限一样，每一个 IPC 对象也具有存取权限，因此，可以把一个 8 进制与掩码或，形成对消息队列的存取权限。

下面我们来创建一个打开或创建消息队列的函数：

```
int open_queue ( key_t keyval )
{
        int     qid;

        if ( (qid = msgget ( keyval, IPC_CREAT | 0660 ) )  == -1)
        {
                return (-1) ;
        }

        return (qid) ;
}
```

注意，这个例子显式地用了 0660 权限。这个函数要么返回一个消息队列的标识号，要么返回-1 而出错。键值作为唯一的参数必须传递给它。

3．系统调用：msgsnd()

一旦我们有了队列识别号，我们就可以在这个队列上执行操作。要把一条消息传递给一个队列，必须用 msgsnd()系统调用。

原型：int msgsnd （ int msqid, struct msgbuf *msgp, int msgsz, int msgflg ）;

返回：成功为 0， 失败为-1。

msgsnd()的第 1 个参数是队列识别号，由 msgget()调用返回。第 2 个参数 msgp 是一个指针，指向我们重新声明和装载的消息缓冲区。msgsz 参数包含了消息以字节为单位的长度，其中包括了消息类型的 4 个字节。

msgflg 参数可以设置成 0（忽略），或者设置或 IPC_NOWAIT ：如果消息队列满，消息不写到队列中，并且控制权返回给调用进程（继续执行）；如果不指定 IPC_NOWAIT，调用进程将挂起（阻塞）直到消息被写到队列中。

下面我们来看一个发送消息的简单函数：

```
int send_message ( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* mymsgbuf 结构的实际长度 */
        length = sizeof (struct ) - sizeof (long) ;

        if ( (result = msgsnd ( qid, qbuf, length, 0) )  == -1)
        {
                return (-1) ;
        }

        return (result) ;
}
```

这个小函数试图把缓冲区 qbuf 中的消息，发送给队列识别号为 qid 的消息队列。

现在，我们在消息队列里有了一条消息，可以用 ipcs 命令来看队列的状态。如何从消息队列检索消息，可以用 msgrcv()系统调用。

4．系统调用：msgrcv()

原型：int msgrcv （ int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg ）;

返回值：成功，则为拷贝到消息缓冲区的字节数，失败为-1。

很明显，第 1 个参数用来指定要检索的队列（必须由 msgget()调用返回），第 2 个参数（msgp）是存放检索到消息的缓冲区的地址，第 3 个参数（msgsz）是消息缓冲区的大小，包括消息类型的长度（4 字节）。

第 4 个参数（mtype）指定了消息的类型。内核将搜索队列中相匹配类型的最早的消息，并且返回这个消息的一个拷贝，返回的消息放在由 msgp 参数指向的地址。这里存在一个特殊的情况，如果传递给 mytype 参数的值为 0，就可以不管类型，只返回队列中最早的消息。

如果传递给参数 msgflg 的值为 IPC_NOWAIT，并且没有可取的消息，那么给调用进程返回 ENOMSG 错误消息，否则，调用进程阻塞，直到一条消息到达队列并且满足 msgrcv()的参数。如果一个客户正在等待消息，而队列被删除，则返回 EIDRM。如果当进程正在阻塞，并且等待一条消息到达但捕获到了一个信号，则返回 EINTR。

下面我们来看一个从我们已建的消息队列中检索消息的例子

```
int read_message ( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* 计算 mymsgbuf 结构的实际大小*/
        length = sizeof (struct mymsgbuf) - sizeof (long) ;

        if ( (result = msgrcv ( qid, qbuf, length, type,  0) )  == -1)
        {
                return (-1) ;
        }

        return (result) ;
}
```

当从队列中成功地检索到消息后，这个消息将从队列中删除。

7.3.3 共享内存

共享内存可以被描述成内存一个区域（段）的映射，这个区域可以被更多的进程所共享。这是 IPC 机制中最快的一种形式，因为它不需要中间环节，而是把信息直接从一个内存段映射到调用进程的地址空间。一个段可以直接由一个进程创建，随后，可以有任意多的进程对其读和写。但是，一旦内存被共享之后，对共享内存的访问同步需要由其他 IPC 机制，例如信号量来实现。像所有的 System V IPC 对象一样，Linux 对共享内存的存取是通过对访问键和访问权限的检查来控制的。

1．数据结构

与消息队列和信号量集合类似，内核为每一个共享内存段（存在于它的地址空间）维护着一个特殊的数据结构 shmid_ds，这个结构在 include/linux/shm.h 中定义如下：

```
/* 在系统中 每一个共享内存段都有一个 shmid_ds 数据结构. */
struct shmid_ds {
    struct ipc_perm shm_perm;         /* 操作权限 */
    int      shm_segsz;                    /* 段的大小（以字节为单位） */
    time_t  shm_atime;                     /* 最后一个进程附加到该段的时间 */
    time_t  shm_dtime;                /* 最后一个进程离开该段的时间 */
    time_t  shm_ctime;                /* 最后一次修改这个结构的时间 */
    unsigned short  shm_cpid;         /*创建该段进程的 pid */
    unsigned short  shm_lpid;         /* 在该段上操作的最后一个进程的 pid */
    short   shm_nattch;               /*当前附加到该段的进程的个数  */

                                           /* 下面是私有的 */

    unsigned short   shm_npages;      /*段的大小（以页为单位） */
   unsigned long   *shm_pages;        /* 指向 frames -> SHMMAX 的指针数组 */
            struct vm_area_struct *attaches;    /* 对共享段的描述 */
    };
```

我们用图 7.4 来表示共享内存的数据结构 shmid_ds 与其他相关数据结构的关系。

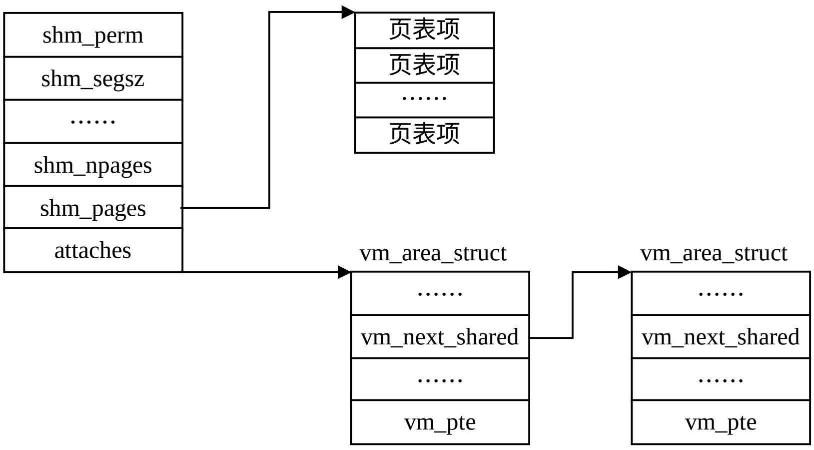


图 7.4　共享内存的数据结构

2．共享内存的处理过程

某个进程第 1 次访问共享虚拟内存时将产生缺页异常。这时，Linux 找出描述该内存的 vm_area_struct 结构，该结构中包含用来处理这种共享虚拟内存段的处理函数地址。共享内存缺页异常处理代码对 shmid_ds 的页表项表进行搜索，以便查看是否存在该共享虚拟内存的页表项。如果没有，系统将分配一个物理页并建立页表项，该页表项加入 shmid_ds 结构的同时也添加到进程的页表中。这就意味着当下一个进程试图访问这页内存时出现缺页异常，共享内存的缺页异常处理代码则把新创建的物理页给这个进程。因此说，第 1 个进程对共享

内存的存取引起创建新的物理页面，而其他进程对共享内存的存取引起把那个页添加到它们的地址空间。

当某个进程不再共享其虚拟内存时，利用系统调用将共享段从自己的虚拟地址区域中移去，并更新进程页表。当最后一个进程释放了共享段之后，系统将释放给共享段所分配的物理页。

当共享的虚拟内存没有被锁定到物理内存时，共享内存也可能会被交换到交换区中。

3．系统调用：shmget()

原型：int shmget （ key_t key，int size，int shmflg ）；

返回：成功，则返回共享内存段的识别号，失败返回-1。

shmget()系统调用类似于信号量和消息队列的系统调用，在此不进一步赘述。

4．系统调用：shmat()

原型：int shmat（int shmid，char *shmaddr，int shmflg）；

返回：成功，则返回附加到进程的那个段的地址，失败返回-1。

其中 shmid 是由 shmget()调用返回的共享内存段识别号，shmaddr 是你希望共享段附加的地址，shmflag 允许你规定希望所附加的段为只读（利用 SHM_RDONLY）以代替读写。通常，并不需要规定你自己的 shmaddr，可以用传递参数值零使得系统为你取得一个地址。

这个调用可能是最简单的，下面看一个例子，把一个有效的识别号传递给一个段，然后返回这个段被附加到内存的内存地址。

```
char *attach_segment ( int shmid )
{
        return (shmat (shmid, 0, 0) ) ;
}
```

一旦一个段适当地被附加，并且一个进程有指向那个段起始地址的一个指针，那么，对那个段的读写就变得相当容易。

5．系统调用：shmctl()

原型： int shmctl （ int shmqid，int cmd，struct shmid_ds *buf ）；

返回：成功返回 0 ，失败返回-1。

这个特殊的调用和 semctl()调用几乎相同，因此，这里不进行详细的讨论。有效命令的值如下所述。

- IPC_STAT：检索一个共享段的 shmid_ds 结构，把它存到 buf 参数的地址中。
- IPC_SET：对一个共享段来说，从 buf 参数中取值设置 shmid_ds 结构的 ipc_perm 域的值。
- IPC_RMID：把一个段标记为删除。
- IPC_RMID 命令实际上不从内核删除一个段，而是仅仅把这个段标记为删除，实际的删除发生在最后一个进程离开这个共享段时。

当一个进程不再需要共享内存段时，它将调用 shmdt()系统调用取消这个段，但是，这并不是从内核真正地删除这个段，而是把相关 shmid_ds 结构的 shm_nattch 域的值减 1，当

这个值为 0 时，内核才从物理上删除这个共享段。

第八章　虚拟文件系统

作为一个最著名的自由软件，Linux 确实名不虚传，几乎处处体现了“自由”，你可以编译适合自己系统要求的内核，或者轻松添加别人开发的新的模块。只要你有实力，你还可以自己写一个新的 Linux 支持的文件系统。写一个新的文件系统虽然是一个“耸人听闻”的事，但 Linux 确实有这样一个特点，就是可以很方便地支持别的操作系统的文件系统，比如 Windows 的文件系统就被 Linux 所支持。Linux 不仅支持多种文件系统，而且还支持这些文件系统相互之间进行访问，这一切都要归功于神奇的虚拟文件系统。

8.1　概述

虚拟文件系统又称虚拟文件系统转换（Virual Filesystem Switch ，简称 VFS）。说它虚拟，是因为它所有的数据结构都是在运行以后才建立，并在卸载时删除，而在磁盘上并没有存储这些数据结构。如果只有 VFS，系统是无法工作的，因为它的这些数据结构不能凭空而来，只有与实际的文件系统，如 Ext2、Minix、MSDOS、VFAT 等相结合，才能开始工作，所以 VFS 并不是一个真正的文件系统。与 VFS 相对应，我们称 Ext2、Minix、MSDOS 等为具体文件系统。

1．虚拟文件系统的作用

在第一章 Linux 内核结构一节中，我们把 VFS 称为内核的一个子系统，其他子系统只与 VFS 打交道，而并不与具体文件系统发生联系。对具体文件系统来说，VFS 是一个管理者，而对内核的其他子系统来说，VFS 是它们与具体文件系统的一个接口，整个 Linux 中文件系统的逻辑关系如图 8.1 所示。

VFS 提供一个统一的接口（实际上就是 file_operatoin 数据结构，稍后介绍），一个具体文件系统要想被 Linux 支持，就必须按照这个接口编写自己的操作函数，而将自己的细节对内核其他子系统隐藏起来。因而，对内核其他子系统以及运行在操作系统之上的用户程序而言，所有的文件系统都是一样的。实际上，要支持一个新的文件系统，主要任务就是编写这些接口函数。

概括说来，VFS 主要有以下几个作用。

（1）对具体文件系统的数据结构进行抽象，以一种统一的数据结构进行管理。

（2）接受用户层的系统调用，例如 write、open、stat、link 等。

（3）支持多种具体文件系统之间相互访问。

（4）接受内核其他子系统的操作请求，特别是内存管理子系统。

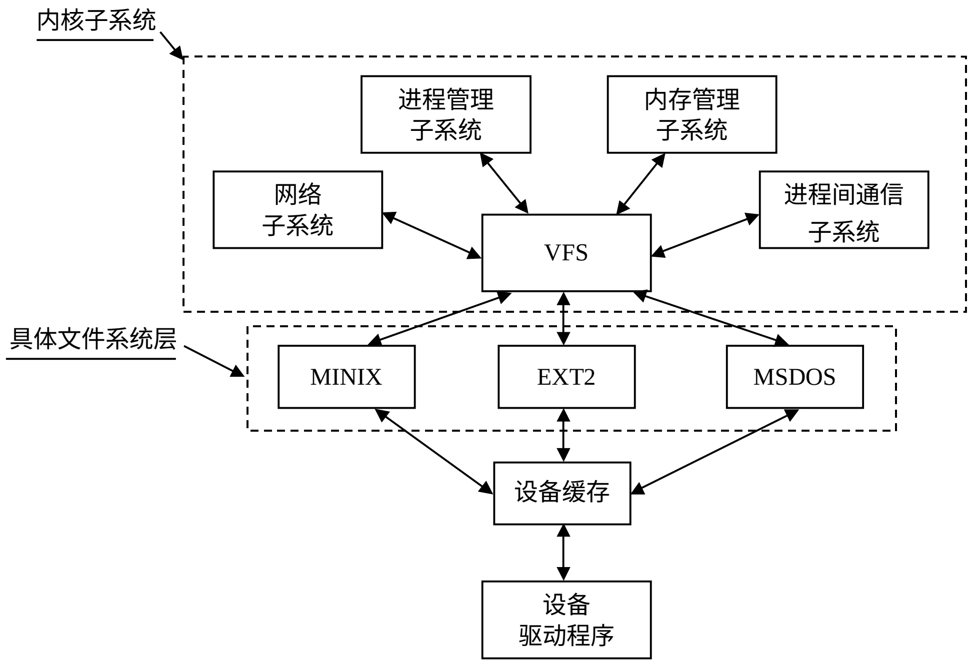


图 8.1　Linux 中文件系统的逻辑关系示意图

通过 VFS，Linux 可以支持很多种具体文件系统，表 8.1 是 Linux 支持的部分具体文件系统。

表 8.1　　Linux 支持的部分文件系统

文件系统	描述
Minix	Linux 最早支持的文件系统。主要缺点是最大 64MB 的磁盘分区和最长 14 个字符的文件名称的限制
Ext	第 1 个 Linux 专用的文件系统，支持 2GB 磁盘分区，255 字符的文件名称，但性能有问题
Xiafs	在 Minix 基础上发展起来，克服了 Minix 的主要缺点。但很快被更完善的文件系统取代
Ext2	当前实际上的 Linux 标准文件系统。性能强大，易扩充，可移植
Ext3	日志文件系统。Ext3 文件系统是对稳定的 Ext2 文件系统的改进
System V	UNIX 早期支持的文件系统，也有与 Minix 同样的限制
NFS	网络文件系统。使得用户可以像访问本地文件一样访问远程主机上的文件
ISO 9660	光盘使用的文件系统
/proc	一个反映内核运行情况的虚的文件系统，并不实际存在于磁盘上
Msdos	DOS 的文件系统，系统力图使它表现得像 UNIX
UMSDOS	该文件系统允许 MSDOS 文件系统可以当作 Linux 固有的文件系统一样使用
Vfat	fat 文件系统的扩展，支持长文件名
Ntfs	Windows NT 的文件系统
Hpfs	OS/2 的文件系统

2．VFS 所处理的系统调用

表 8.2 列出 VFS 的系统调用，这些系统调用涉及文件系统、常规文件、目录及符号链接。另外还有少数几个由 VFS 处理的其他系统调用：诸如 ioperm( )、ioctl( )、pipe( )和 mknod( )，涉及设备文件和管道文件，有些内容在下一章进行讨论。由 VFS 处理的最后一组系统调用，诸如 socket( )、connect( )、bind( )和 protocols( )，属于套接字系统调用并用于实现网络功能。

表 8.2　　VFS 的部分系统调用

系统调用名	功能
mount( )/ umount( )	安装/卸载文件系统
sysfs( )	获取文件系统信息
statfs( )/ fstatfs( ) /ustat( )	获取文件系统统计信息
chroot( )	更改根目录
chdir( ) /fchdir( ) /getcwd( )	更改当前目录
mkdir( ) /rmdir( )	创建/删除目录
getdents( ) /readdir( )/ link( ) unlink( ) /rename( )	对目录项进行操作
readlink( ) /symlink( )	对软链接进行操作
chown( ) /fchown( ) /lchown( )	更改文件所有者
chmod( )/ fchmod( ) /utime( )	更改文件属性
stat( ) /fstat( ) /lstat( ) access( )	读取文件状态
open( ) /close( ) /creat( ) /umask( )	打开/关闭文件
dup( ) /dup2( ) /fcntl( )	对文件描述符进行操作
select( ) /poll( )	异步 I/O 通信
truncate( ) /ftruncate( )	更改文件长度
lseek( ) /_llseek( )	更改文件指针
read( )/ write( ) /readv( ) /writev( ) sendfile( )	文件 I/O 操作
pread( )/ pwrite( )	搜索并访问文件
mmap( ) /munmap( )	文件内存映射
fdatasync( ) /fsync( ) /sync( )/ msync( )	同步访问文件数据
flock( )	处理文件锁

前面我们已经提到，VFS 是应用程序和具体的文件系统之间的一个层。不过，在某些情

况下，一个文件操作可能由 VFS 本身去执行，无需调用下一层程序。例如，当某个进程关闭一个打开的文件时，并不需要涉及磁盘上的相应文件，因此，VFS 只需释放对应的文件对象。类似地，如果系统调用 lseek()修改一个文件指针，而这个文件指针指向有关打开的文件与进程交互的一个属性，那么 VFS 只需修改对应的文件对象，而不必访问磁盘上的文件，因此，无需调用具体的文件系统子程序。从某种意义上说，可以把 VFS 看成“通用”文件系统，它在必要时依赖某种具体的文件系统。

8.2 VFS 中的数据结构

虚拟文件系统所隐含的主要思想在于引入了一个通用的文件模型，这个模型能够表示所有支持的文件系统。该模型严格遵守传统 UNIX 文件系统提供的文件模型。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux 的编码并未采用面向对象的程序设计语言（比如 C++）。因此对象作为数据结构来实现：数据结构中指向函数的域就对应于对象的方法。

通用文件模型由下列对象类型组成。

- 超级块（superblock）对象：存放系统中已安装文件系统的有关信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块，也就是说，每个文件系统都有一个超级块对象。
- 索引节点（inode）对象：存放关于具体文件的一般信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块（FCB），也就是说，每个文件都有一个索引节点对象。每个索引节点对象都有一个索引节点号，这个号唯一地标识某个文件系统中的指定文件。
- 目录项（dentry）对象：存放目录项与对应文件进行链接的信息。VFS 把每个目录看作一个由若干子目录和文件组成的常规文件。例如，在查找路径名/tmp/test 时，内核为根目录“/”创建一个目录项对象，为根目录下的 tmp 项创建一个第 2 级目录项对象，为/tmp 目录下的 test 项创建一个第 3 级目录项对象。
- 文件（file）对象：存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内存中。

下面我们讨论超级块、索引节点、目录项及文件的数据结构，它们的共同特点有两个：

- 充分考虑到对多种具体文件系统的兼容性；
- 是“虚”的，也就是说只能存在于内存。

这正体现了 VFS 的特点，在下面的描述中，读者也许能体会到以上特点。

8.2.1 超级块

很多具体文件系统中都有超级块结构，超级块是这些文件系统中最重要的数据结构，它是来描述整个文件系统信息的，可以说是一个全局的数据结构。Minix、Ext2 等有超级块，

VFS 也有超级块，为了避免与后面介绍的 Ext2 超级块发生混淆，这里用 VFS 超级块来表示。VFS 超级块是各种具体文件系统在安装时建立的，并在这些文件系统卸载时自动删除，可见，VFS 超级块确实只存在于内存中,同时提到 VFS 超级块也应该说成是哪个具体文件系统的 VFS 超级块。VFS 超级块在 inculde/fs/fs.h 中定义，即数据结构 super_block，该结构及其主要域的含义如下：

```
struct super_block
{
/************描述具体文件系统的整体信息的域*****************
       kdev_t s_dev;                 /* 包含该具体文件系统的块设备标识符。
例如，对于 /dev/hda1，其设备标识符为 0x301*/
     unsigned long s_blocksize;         /*该具体文件系统中数据块的大小，
以字节为单位 */
     unsigned char s_blocksize_bits;    /*块大小的值占用的位数，例如，
如果块大小为 1024 字节，则该值为 10*/
       unsigned long long       s_maxbytes;      /* 文件的最大长度 */
     unsigned long s_flags;            /* 安装标志*/
     unsigned long s_magic;            /*魔数，即该具体文件系统区别于其他
    文件系统的一个标志*/

/**************用于管理超级块的域******************/
    struct list_head    s_list;    /*指向超级块链表的指针*/
struct semaphore      s_lock       /*锁标志位，若置该位，则其他进程
                            不能对该超级块操作*/
    struct rw_semaphore      s_umount    /*对超级块读写时进行同步*/
unsigned char s_dirt;         /*脏位，若置该位，表明该超级块已被修改*/
struct dentry      *s_root; /*指向该具体文件系统安装目录的目录项*/
int                        s_count; /*对超级块的使用计数*/
atomic_t                   s_active;
struct list_head        s_dirty;         /*已修改的索引节点形成的链表 */
struct list_head         s_locked_inodes;/* 要进行同步的索引节点形成的链表*/
struct list_head         s_files
/***********和具体文件系统相联系的域*************************/
     struct file_system_type *s_type;                /*指向文件系统的
                                         file_system_type 数据结构的指针 */
     struct super_operations *s_op;             /*指向某个特定的具体文件系统的用
                                        于超级块操作的函数集合 */
struct dquot_operations *dq_op;               /* 指向某个特定的具体文件系统
                                         用于限额操作的函数集合 */
u;                                  /*一个共用体，其成员是各种文件系统
                                       的 fsname_sb_info 数据结构 */
};
```

所有超级块对象（每个已安装的文件系统都有一个超级块）以双向环形链表的形式链接在一起。链表中第一个元素和最后一个元素的地址分别存放在 super_blocks 变量的 s_list 域的 next 和 prev 域中。s_list 域的数据类型为 struct list_head，在超级块的 s_dirty 域以及内核的其他很多地方都可以找到这样的数据类型 ;这种数据类型仅仅包括指向链表中的前一个元素和后一个元素的指针。因此，超级块对象的 s_list 域包含指

向链表中两个相邻超级块对象的指针。图 8.2 说明了 list_head 元素、next 和 prev 是如何嵌入到超级块对象中的。

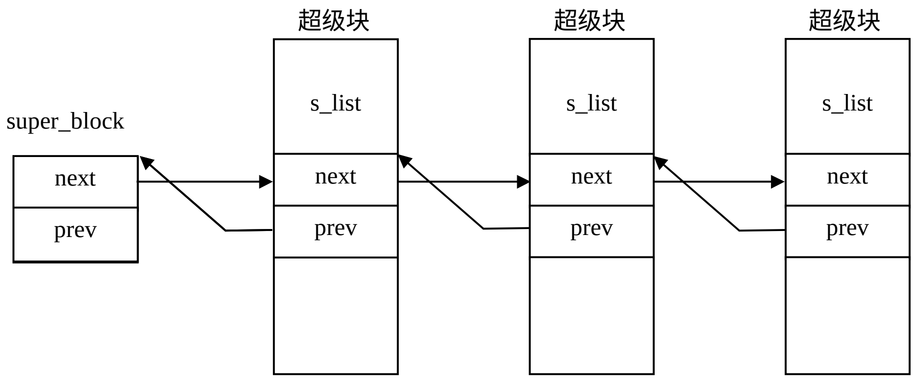


图 8.2　超级块链表

超级块最后一个 u 联合体域包括属于具体文件系统的超级块信息：

```
        union {
                struct Minix_sb_info    Minix_sb;
                struct Ext2_sb_info     Ext2_sb;
                struct ext3_sb_info     ext3_sb;
                struct hpfs_sb_info     hpfs_sb;
                struct ntfs_sb_info     ntfs_sb;
                struct msdos_sb_info    msdos_sb;
                struct isofs_sb_info    isofs_sb;
                struct nfs_sb_info      nfs_sb;
                struct sysv_sb_info     sysv_sb;
                struct affs_sb_info     affs_sb;
                struct ufs_sb_info      ufs_sb;
                struct efs_sb_info      efs_sb;
                struct shmem_sb_info    shmem_sb;
                struct romfs_sb_info    romfs_sb;
                struct smb_sb_info      smbfs_sb;
                struct hfs_sb_info      hfs_sb;
                struct adfs_sb_info     adfs_sb;
                struct qnx4_sb_info     qnx4_sb;
                struct reiserfs_sb_info reiserfs_sb;
                struct bfs_sb_info      bfs_sb;
                struct udf_sb_info      udf_sb;
               struct ncp_sb_info      ncpfs_sb;
                struct usbdev_sb_info   usbdevfs_sb;
               struct jffs2_sb_info    jffs2_sb;
               struct cramfs_sb_info   cramfs_sb;
                void                    *generic_sbp;
        } u;
```

通常，为了效率起见 u 域的数据被复制到内存。任何基于磁盘的文件系统都需要访问和更改自己的磁盘分配位示图，以便分配和释放磁盘块。VFS 允许这些文件系统直接对内存超

级块的 u 联合体域进行操作，无需访问磁盘。

但是，这种方法带来一个新问题：有可能 VFS 超级块最终不再与磁盘上相应的超级块同步。因此，有必要引入一个 s_dirt 标志，来表示该超级块是否是脏的，也就是说，磁盘上的数据是否必须要更新。缺乏同步还导致我们熟悉的一个问题：当一台机器的电源突然断开而用户来不及正常关闭系统时，就会出现文件系统崩溃。Linux 是通过周期性地将所有“脏”的超级块写回磁盘来减少该问题带来的危害。

与超级块关联的方法就是所谓的超级块操作。这些操作是由数据结构 super_operations 来描述的，该结构的起始地址存放在超级块的 s_op 域中，稍后将与其他对象的操作一块儿介绍。

8.2.2　VFS 的索引节点

文件系统处理文件所需要的所有信息都放在称为索引节点的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件的存在而存在。有关使用索引节点的原因将在下一章中进一步介绍，这里主要强调一点，具体文件系统的索引节点是存储在磁盘上的，是一种静态结构，要使用它，必须调入内存，填写 VFS 的索引节点，因此，也称 VFS 索引节点为动态节点。这里用 VFS 索引节点来避免与下一章的 Ext2 索引节点混淆。VFS 索引节点的数据结构 inode 在/includ/fs/fs.h 中定义如下（2.4.x 版本）：

```
  struct inode
{
     /**********描述索引节点高速缓存管理的域****************/
 struct list_head        i_hash;  ／*指向哈希链表的指针*/
 struct list_head        i_list;  /*指向索引节点链表的指针*/
 struct list_head        i_dentry;/*指向目录项链表的指针*/

 struct list_head        i_dirty_buffers;
 struct list_head        i_dirty_data_buffers;
 /**********描述文件信息的域****************/
    unsigned long i_ino;            /*索引节点号*/
    kdev_t         i_dev;           /*设备标识号 */
  umode_t      i_mode;          /*文件的类型与访问权限 */
  nlink_t      i_nlink;          /*与该节点建立链接的文件数 */
    uid_t          i_uid;             /*文件拥有者标识号*/
    gid_t          i_gid;              /*文件拥有者所在组的标识号*/
     kdev_t           i_rdev;             /*实际设备标识号*/
    off_t          i_size;             /*文件的大小（以字节为单位）*/
    unsigned long i_blksize;         /*块大小*/
    unsigned long i_blocks;          /*该文件所占块数*/
  time_t       i_atime;          /*文件的最后访问时间*/
    time_t         i_mtime;          /*文件的最后修改时间*/
    time_t         i_ctime;           /*节点的修改时间*/
    unsigned long i_version;         /*版本号*/
  struct semaphore   i_zombie;   /*僵死索引节点的信号量*/

/***********用于索引节点操作的域*****************/
```

```
    struct inode_operations  *i_op;  /*索引节点的操作*/
    struct super_block   *i_sb;     /*指向该文件系统超级块的指针 */
        atomic_t     i_count;     /*当前使用该节点的进程数。计数为 0，
    表明该节点可丢弃或被重新使用 */
      struct file_operations  *i_fop;  /*指向文件操作的指针 */

       unsigned char      i_lock;           /*该节点是否被锁定，用于同步操作中*/
     struct semaphore     i_sem;           /*指向用于同步操作的信号量结构*/
     wait_queue_head_t   *i_wait;         /*指向索引节点等待队列的指针*/
     unsigned char          i_dirt;         /*表明该节点是否被修改过，若已被修改，
                            则应当将该节点写回磁盘*/
    struct file_lock       *i_flock;      /*指向文件加锁链表的指针*/
    struct dquot           *i_dquot[MAXQUOTAS]; /*索引节点的磁盘限额*/
    /************用于分页机制的域**********************************/
    struct address_space    *i_mapping; ／* 把所有可交换的页面管理起来*/
         struct address_space    i_data;

    /**********以下几个域应当是联合体****************************************/
       struct list_head        i_devices;   /*设备文件形成的链表*/
       struct pipe_inode_info  i_pipe;     /*指向管道文件*/
       struct block_device     *i_bdev;   /*指向块设备文件的指针*/
       struct char_device      *i_cdev; /*指向字符设备文件的指针*/

    /************************其他域**************************************/
    unsigned long           i_dnotify_mask; /* Directory notify events */
    struct dnotify_struct   *i_dnotify; /* for directory notifications */

    unsigned long           i_state;    /*索引节点的状态标志*/
    unsigned int            i_flags;    /*文件系统的安装标志*/
    unsigned char           i_sock;     /*如果是套接字文件则为真*/
    atomic_t                i_writecount; /*写进程的引用计数*/
    unsigned int            i_attr_flags; /*文件创建标志*/
    __u32                   i_generation  /*为以后的开发保留*/
    /************************各个具体文件系统的索引节点********************/
    union；                           /*类似于超级块的一个共用体，其成员是各种具体文件系统
的 fsname_inode_info 数据结构 */
    }
```

对 inode 数据结构的进一步说明。

- 每个文件都有一个 inode，每个 inode 有一个索引节点号 i_ino。在同一个文件系统中，每个索引节点号都是唯一的，内核有时根据索引节点号的哈希值查找其 inode 结构。
- 每个文件都有个文件主，其最初的文件主是创建了这个文件的用户，但以后可以改变。每个用户都有一个用户组，且属于某个用户组，因此，inode 结构中就有相应的 i_uid、i_gid，以指明文件主的身份。
- inode 中有两个设备号，i_dev 和 i_rdev。首先，除特殊文件外，每个节点都存储在某个设备上，这就是 i_dev。其次，如果索引节点所代表的并不是常规文件，而是某个设备，那就还得有个设备号，这就是 i_rdev。
- 每当一个文件被访问时，系统都要在这个文件的 inode 中记下时间标记，这就是 inode

中与时间相关的几个域。

- 每个索引节点都会复制磁盘索引节点包含的一些数据，比如文件占用的磁盘块数。如果 i_state 域的值等于 I_DIRTY，该索引节点就是“脏”的，也就是说，对应的磁盘索引节点必须被更新。i_state 域的其他值有 I_LOCK（这意味着该索引节点对象已加锁），I_FREEING（这意味着该索引节点对象正在被释放）。每个索引节点对象总是出现在下列循环双向链表的某个链表中。

（1）未用索引节点链表。变量 inode_unused 的 next 域和 prev 域分别指向该链表中的首元素和尾元素。这个链表用做内存高速缓存。

（2）正在使用索引节点链表。变量 inode_in_use 指向该链表中的首元素和尾元素。

（3）脏索引节点链表。由相应超级块的 s_dirty 域指向该链表中的首元素和尾元素。

这 3 个链表都是通过索引节点的 i_list 域链接在一起的。

- 属于“正在使用”或“脏”链表的索引节点对象也同时存放在一个称为 inode_hashtable 链表中。哈希表加快了对索引节点对象的搜索，前提是系统内核要知道索引节点号及对应文件所在文件系统的超级块对象的地址。由于散列技术可能引发冲突，所以，索引节点对象设置一个 i_hash 域，其中包含向前和向后的两个指针，分别指向散列到同一地址的前一个索引节点和后一个索引节点；该域由此创建了由这些索引节点组成的一个双向链表。

与索引节点关联的方法也叫索引节点操作，由 inode_operations 结构来描述，该结构的地址存放在 i_op 域中，该结构也包括一个指向文件操作方法的指针。

8.2.3　目录项对象

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry（directory enrty）数据结构。dentry 结构中有个 d_inode 指针指向相应的 inode 结构。读者也许会问，既然 inode 结构和 dentry 结构都是对文件各方面属性的描述，那为什么不把这两个结构“合二为一”呢？这是因为二者所描述的目标不同，dentry 结构代表的是逻辑意义上的文件，所描述的是文件逻辑上的属性，因此，目录项对象在磁盘上并没有对应的映像；而 inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统（如 Ext2），Ext2_ inode 结构在磁盘上就有对应的映像。所以说，一个索引节点对象可能对应多个目录项对象。

dentry 的定义在 include/linux/dcache.h 中：

```
struct dentry {
    atomic_t d_count;         /* 目录项引用计数器 */
    unsigned int d_flags;     /* 目录项标志 */
    struct inode  * d_inode;   /* 与文件名关联的索引节点 */
    struct dentry * d_parent;        /* 父目录的目录项 */
    struct list_head d_hash;         /* 目录项形成的哈希表 */
    struct list_head d_lru;          /*未使用的 LRU 链表 */
   struct list_head d_child;        /*父目录的子目录项所形成的链表 */
  struct list_head d_subdirs;     /* 该目录项的子目录所形成的链表*/
  struct list_head d_alias;       /* 索引节点别名的链表*/
```

```
        int d_mounted;                  /* 目录项的安装点 */
        struct qstr d_name;             /* 目录项名（可快速查找） */
        unsigned long d_time;           /* 由 d_revalidate 函数使用 */
        struct dentry_operations  *d_op; /* 目录项的函数集*/
        struct super_block * d_sb;      /* 目录项树的根 （即文件的超级块）*/
        unsigned long d_vfs_flags;
        void * d_fsdata;                /* 具体文件系统的数据 */
        unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
};
```

下面对 dentry 结构给出进一步的解释。

一个有效的 dentry 结构必定有一个 inode 结构，这是因为一个目录项要么代表着一个文件，要么代表着一个目录，而目录实际上也是文件。所以，只要 dentry 结构是有效的，则其指针 d_inode 必定指向一个 inode 结构。可是，反过来则不然，一个 inode 却可能对应着不止一个 dentry 结构；也就是说，一个文件可以有不止一个文件名或路径名。这是因为一个已经建立的文件可以被连接(link)到其他文件名。所以在 inode 结构中有一个队列 i_dentry，凡是代表着同一个文件的所有目录项都通过其 dentry 结构中的 d_alias 域挂入相应 inode 结构中的 i_dentry 队列。

在内核中有一个哈希表 dentry_hashtable ，是一个 list_head 的指针数组。一旦在内存中建立起一个目录节点的 dentry 结构，该 dentry 结构就通过其 d_hash 域链入哈希表中的某个队列中。

内核中还有一个队列 dentry_unused，凡是已经没有用户（count 域为 0）使用的 dentry 结构就通过其 d_lru 域挂入这个队列。

Dentry 结构中除了 d_alias、d_hash、d_lru 三个队列外，还有 d_vfsmnt、d_child 及 d_subdir 三个队列。其中 d_vfsmnt 仅在该 dentry 为一个安装点时才使用。另外，当该目录节点有父目录时，则其 dentry 结构就通过 d_child 挂入其父节点的 d_subdirs 队列中，同时又通过指针 d_parent 指向其父目录的 dentry 结构,而它自己各个子目录的 dentry 结构则挂在其 d_subdirs 域指向的队列中。

从上面的叙述可以看出，一个文件系统中所有目录项结构或组织为一个哈希表，或组织为一颗树，或按照某种需要组织为一个链表，这将为文件访问和文件路径搜索奠定下良好的基础。

8.2.4　与进程相关的文件结构

在具体介绍这几个结构以前，我们需要解释一下文件描述符、打开的文件描述、系统打开文件表、用户打开文件表的概念以及它们的联系。

1．文件对象

在 Linux 中，进程是通过文件描述符（file descriptors，简称 fd）而不是文件名来访问文件的，文件描述符实际上是一个整数。Linux 中规定每个进程最多能同时使用 NR_OPEN 个文件描述符，这个值在 fs.h 中定义，为 1024×1024（2.0 版中仅定义为 256）。

每个文件都有一个 32 位的数字来表示下一个读写的字节位置，这个数字叫做文件位置。

每次打开一个文件，除非明确要求，否则文件位置都被置为 0，即文件的开始处，此后的读或写操作都将从文件的开始处执行，但你可以通过执行系统调用 LSEEK（随机存储）对这个文件位置进行修改。Linux 中专门用了一个数据结构 file 来保存打开文件的文件位置，这个结构称为打开的文件描述（open file description）。这个数据结构的设置是煞费苦心的，因为它与进程的联系非常紧密，可以说这是 VFS 中一个比较难于理解的数据结构。

首先，为什么不把文件位置干脆存放在索引节点中，而要多此一举，设一个新的数据结构呢？我们知道，Linux 中的文件是能够共享的，假如把文件位置存放在索引节点中，则如果有两个或更多个进程同时打开同一个文件时，它们将去访问同一个索引节点，于是一个进程的 LSEEK 操作将影响到另一个进程的读操作，这显然是不允许也是不可想象的。

另一个想法是既然进程是通过文件描述符访问文件的，为什么不用一个与文件描述符数组相平行的数组来保存每个打开文件的文件位置？这个想法也是不能实现的，原因就在于在生成一个新进程时，子进程要共享父进程的所有信息，包括文件描述符数组。

我们知道，一个文件不仅可以被不同的进程分别打开，而且也可以被同一个进程先后多次打开。一个进程如果先后多次打开同一个文件，则每一次打开都要分配一个新的文件描述符，并且指向一个新的 file 结构，尽管它们都指向同一个索引节点，但是，如果一个子进程不和父进程共享同一个 file 结构，而是也如上面一样，分配一个新的 file 结构，会出现什么情况了？让我们来看一个例子。

假设有一个输出重定位到某文件 A 的 shell script（shell 脚本），我们知道，shell 是作为一个进程运行的，当它生成第 1 个子进程时，将以 0 作为 A 的文件位置开始输出，假设输出了 2KB 的数据，则现在文件位置为 2KB。然后，shell 继续读取脚本，生成另一个子进程，它要共享 shell 的 file 结构，也就是共享文件位置，所以第 2 个进程的文件位置是 2KB，将接着第 1 个进程输出内容的后面输出。如果 shell 不和子进程共享文件位置，则第 2 个进程就有可能重写第 1 个进程的输出了，这显然不是希望得到的结果。

至此，已经可以看出设置 file 结构的原因所在了。

file 结构中主要保存了文件位置，此外，还把指向该文件索引节点的指针也放在其中。file 结构形成一个双链表，称为系统打开文件表，其最大长度是 NR_FILE，在 fs.h 中定义为 8192。

file 结构在 include\linux\fs.h 中定义如下：

```
struct file
{
 struct list_head        f_list;     /*所有打开的文件形成一个链表*/
 struct dentry           *f_dentry;  /*指向相关目录项的指针*/
 struct vfsmount         *f_vfsmnt;  /*指向 VFS 安装点的指针*/
 struct file_operations  *f_op;      /*指向文件操作表的指针*/
  mode_t f_mode;                                  /*文件的打开模式*/
  loff_t f_pos;                                   /*文件的当前位置*/
     unsigned short f_flags;                         /*打开文件时所指定的标志*/
     unsigned short f_count;                           /*使用该结构的进程数*/
     unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
   /*预读标志、要预读的最多页面数、上次预读后的文件指针、预读的字节数以及
    预读的页面数*/
  int f_owner;                    /* 通过信号进行异步 I/O 数据的传送*/
```

```
    unsigned int          f_uid, f_gid;   /*用户的 UID 和 GID*/
    int                  f_error;        /*网络写操作的错误码*/

    unsigned long f_version;             /*版本号*/
      void *private_data;                          /* tty 驱动程序所需 */

};
```

每个文件对象总是包含在下列的一个双向循环链表之中。

- “未使用”文件对象的链表。该链表既可以用做文件对象的内存高速缓存，又可以当作超级用户的备用存储器，也就是说，即使系统的动态内存用完，也允许超级用户打开文件。由于这些对象是未使用的，它们的 f_count 域是 NULL，该链表首元素的地址存放在变量 free_list 中，内核必须确认该链表总是至少包含 NR_RESERVED_FILES 个对象，通常该值设为 10。
- “正在使用”文件对象的链表。该链表中的每个元素至少由一个进程使用，因此，各个元素的 f_count 域不会为 NULL，该链表中第一个元素的地址存放在变量 anon_list 中。

如果 VFS 需要分配一个新的文件对象，就调用函数 get_empty_filp( )。该函数检测“未使用”文件对象链表的元素个数是否多于 NR_RESERVED_FILES，如果是，可以为新打开的文件使用其中的一个元素；如果没有，则退回到正常的内存分配。

2．用户打开文件表

每个进程用一个 files_struct 结构来记录文件描述符的使用情况，这个 files_struct 结构称为用户打开文件表，它是进程的私有数据。files_struct 结构在 include/linux/sched.h 中定义如下：

```
struct files_struct {
     atomic_t count;         /* 共享该表的进程数 */
     rwlock_t file_lock;      /* 保护以下的所有域,以免在
tsk->alloc_lock 中的嵌套*/
     int max_fds;             /*当前文件对象的最大数*/
     int max_fdset;          /*当前文件描述符的最大数*/
          int next_fd;             ／*已分配的文件描述符加 1*/
      struct file ** fd;       /* 指向文件对象指针数组的指针 */
      fd_set *close_on_exec;   /*指向执行 exec（ ）时需要关闭的文件描述符*/
      fd_set *open_fds;       /*指向打开文件描述符的指针*/
       fd_set close_on_exec_init;/* 执行 exec（ ）时需要关闭的文件描述符的初                值
集合*/
          fd_set open_fds_init;  /*文件描述符的初值集合*/
      struct file * fd_array[32];/* 文件对象指针的初始化数组*/
};
```

fd 域指向文件对象的指针数组。该数组的长度存放在 max_fds 域中。通常，fd 域指向 files_struct 结构的 fd_array 域，该域包括 32 个文件对象指针。如果进程打开的文件数目多于 32，内核就分配一个新的、更大的文件指针数组，并将其地址存放在 fd 域中；内核同时也更新 max_fds 域的值。

对于在 fd 数组中有入口地址的每个文件来说，数组的索引就是文件描述符（file descriptor)。通常，数组的第 1 个元素（索引为 0）是进程的标准输入文件，数组的第 2 个

元素（索引为 1）是进程的标准输出文件，数组的第 3 个元素（索引为 2）是进程的标准错误文件（参见图 8.3)。请注意，借助于 dup()、dup2()和 fcntl() 系统调用，两个文件描述符就可以指向同一个打开的文件，也就是说，数组的两个元素可能指向同一个文件对象。当用户使用 shell 结构（如 2>&1）将标准错误文件重定向到标准输出文件上时，用户总能看到这一点。

open_fds 域包含 open_fds_init 域的地址，open_fds_init 域表示当前已打开文件的文件描述符的位图。max_fdset 域存放位图中的位数。由于数据结构 fd_set 有 1024 位，通常不需要扩大位图的大小。不过，如果确实需要，内核仍能动态增加位图的大小，这非常类似文件对象的数组的情形。

当开始使用一个文件对象时调用内核提供的 fget()函数。这个函数接收文件描述符 fd 作为参数，返回在 current->files->fd[fd]中的地址，即对应文件对象的地址，如果没有任何文件与 fd 对应，则返回 NULL。在第 1 种情况下，fget()使文件对象引用计数器 f_count 的值增 1。

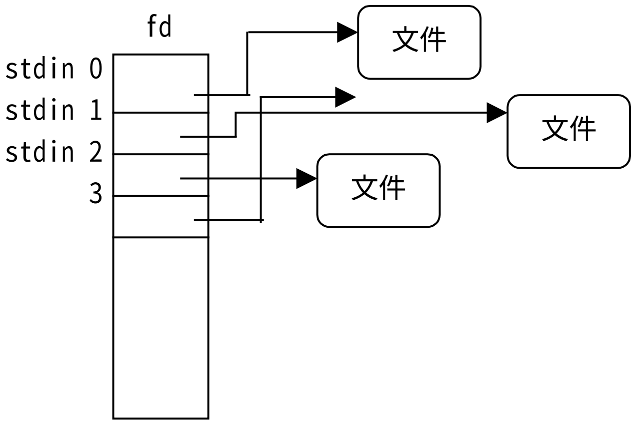


图 8.3　文件描述符数组

当内核完成对文件对象的使用时，调用内核提供的 fput() 函数。该函数将文件对象的地址作为参数，并递减文件对象引用计数器 f_count 的值，另外，如果这个域变为 NULL，该函数就调用文件操作的“释放”方法（如果已定义），释放相应的目录项对象，并递减对应索引节点对象的 i_writeaccess 域的值（如果该文件是写打开），最后，将该文件对象从“正在使用”链表移到“未使用”链表。

3．关于文件系统信息的 fs_struct 结构

fs_struct 结构在 2.4 以前的版本中在 include/linux/sched.h 中定义为：

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd;
};
```

在 2.4 版本中，单独定义在 include/linux/fs_struct.h 中:

```
struct fs_struct {
        atomic_t count;
        rwlock_t lock;
        int umask;
        struct dentry * root, * pwd, * altroot;
        struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

count 域表示共享同一 fs_struct 表的进程数目。umask 域由 umask（）系统调用使用，用于为新创建的文件设置初始文件许可权。

fs_struct 中的 dentry 结构是对一个目录项的描述，root、pwd 及 altroot 三个指针都指向这个结构。其中，root 所指向的 dentry 结构代表着本进程所在的根目录，也就是在用户登录进入系统时所看到的根目录；pwd 指向进程当前所在的目录；而 altroot 则是为用户设置的替换根目录。实际运行时，这 3 个目录不一定都在同一个文件系统中。例如，进程的根目录通常是安装于“／”节点上的 Ext2 文件系统，而当前工作目录可能是安装于／msdos 的一个 DOS 文件系统。因此，fs_struct 结构中的 rootmnt、 pwdmnt 及 altrootmnt 就是对那 3 个目录的安装点的描述，安装点的数据结构为 vfsmount。

8.2.5 主要数据结构间的关系

前面我们介绍了超级块对象、索引节点对象、文件对象及目录项对象的数据结构。我们在此给出这些数据结构之间的联系。

超级块是对一个文件系统的描述；索引节点是对一个文件物理属性的描述；而目录项是对一个文件逻辑属性的描述。除此之外，文件与进程之间的关系是由另外的数据结构来描述的。一个进程所处的位置是由 fs_struct 来描述的，而一个进程（或用户）打开的文件是由 files_struct 来描述的，而整个系统所打开的文件是由 file 结构来描述。如图 8.4 给出了这些数据结构之间的关系。

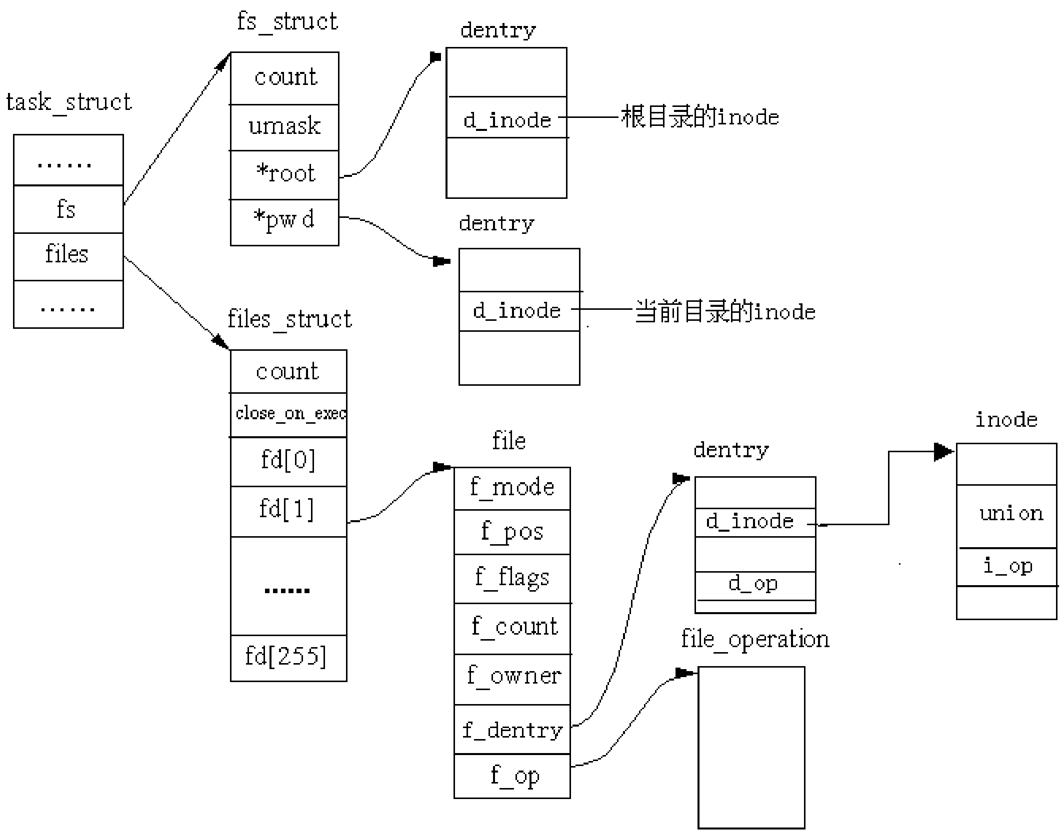


图 8.4　与进程联系的文件结构的关系示意图

8.2.6　有关操作的数据结构

VFS 毕竟是虚拟的，它无法涉及到具体文件系统的细节，所以必然在 VFS 和具体文件系统之间有一些接口，这就是 VFS 设计的一些有关操作的数据结构。这些数据结构就好像是一个标准，具体文件系统要想被 Linux 支持，就必须按这个标准来编写自己操作函数。实际上，也正是这样，各种 Linux 支持的具体文件系统都有一套自己的操作函数，在安装时，这些结构体的成员指针将被初始化，指向对应的函数。如果说 VFS 体现了 Linux 的优越性，那么这些数据结构的设计就体现了 VFS 的优越性所在。图 8.5 是 VFS 和具体文件系统的关系示意图。

这个示意图还无法完全反映这种关系。因为对每个具体文件系统来说，VFS 都有相应的数据结构对应，而不是如图中那样简单化了。

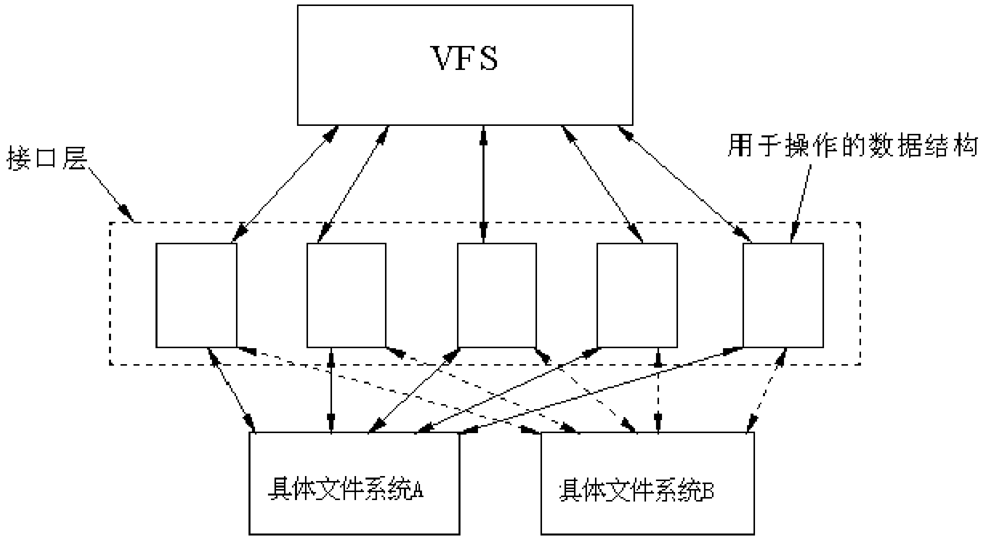


图 8.5　VFS 和具体文件系统的关系示意图

有关操作的数据结构主要有以下几个，分别用来操作 VFS 中的几个重要的数据结构。

1．超级块操作

超级块操作是由 super_operations 数据结构来描述的，该结构的起始地址存放在超级块的 s_op 域中。该结构定义于 fs.h 中：

```
/*
* NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
* without the big kernel lock held in all filesystems.
*/
struct super_operations {
        void  (*read_inode)   (struct inode *) ;
        void  (*read_inode2)   (struct inode *, void *)   ;
        void  (*dirty_inode)   (struct inode *) ;
        void  (*write_inode)   (struct inode *, int) ;
        void  (*put_inode)   (struct inode *) ;
        void  (*delete_inode)   (struct inode *) ;
        void  (*put_super)   (struct super_block *) ;
        void  (*write_super)   (struct super_block *) ;
        void  (*write_super_lockfs)   (struct super_block *) ;
        void  (*unlockfs)   (struct super_block *) ;
        int  (*statfs)   (struct super_block *, struct statfs *) ;
        int  (*remount_fs)   (struct super_block *, int *, char *) ;
        void  (*clear_inode)   (struct inode *) ;
        void  (*umount_begin)   (struct super_block *) ;
}
```

其中的每个函数就叫做超级块的一个方法，表 8.3 给予描述。

表 8.3　　　　　　　　　　　超级块对象的方法及其描述

函数形式	描述

Read_inode(inode)	inode 的地址是该函数的参数，inode 中的 i_no 域表示从磁盘要读取的具体文件系统的 inode，用磁盘上的数据填充参数 inode 的域
Dirty_inode(inode)	把 inode 标记为“脏”
Write_inode(inode)	用参数指定的 inode 更新某个文件系统的 inode。inode 的 i_ino 域标识指定磁盘上文件系统的索引节点
Put_inode(inode)	释放参数指定的索引节点对象。释放一个对象并不意味着释放内存，因为其他进程可能仍然在使用这个对象。该方法是可选的（即并非所有的文件系统都有相应的处理函数）
Delete_inode(inode)	删除那些包含文件、磁盘索引节点及 VFS 索引节点的数据块
Notify_change(dentry, iattr)	依照参数 iattr 的值修改索引节点的一些属性。如果没有定义该函数，VFS 转去执行 write_inode( ) 方法
Put_super(super)	释放超级块对象
Write_super(super)	将超级块的信息写回磁盘，该方法是可选的
Statfs(super, buf, bufsize)	将文件系统的统计信息填写在 buf 缓冲区中
Remount_fs(super, flags, data)	用新的选项重新安装文件系统（当某个安装选项必须被修改时进行调用）
Clear_inode(inode)	与 put_inode 类似，但同时也把索引节点对应文件中的数据占用的所有页释放
Umount_begin(super)	中断一个安装操作（只在网络文件系统中使用）

上面这些方法对所有的文件系统都是适用的，但对于一个具体的文件系统来说，可能只用到其中的几个方法。如果那些方法没有定义，则对应的域为空。

2．索引节点操作 inode_operations

索引节点操作是由 inode_operations 结构来描述的，主要是用来将 VFS 对索引节点的操作转化为具体文件系统处理相应操作的函数，在 fs.h 中描述如下：

```
struct inode_operations {
        int  (*create)  (struct inode *,struct dentry *,int) ;
        struct dentry *  (*lookup)  (struct inode *,struct dentry *) ;
        int  (*link)  (struct dentry *,struct inode *,struct dentry *) ;
        int  (*unlink)  (struct inode *,struct dentry *) ;
        int  (*symlink)  (struct inode *,struct dentry *,const char *) ;
        int  (*mkdir)  (struct inode *,struct dentry *,int) ;
        int  (*rmdir)  (struct inode *,struct dentry *) ;
        int  (*mknod)  (struct inode *,struct dentry *,int,int) ;
        int  (*rename)  (struct inode *, struct dentry *,
                        struct inode *, struct dentry *) ;
        int  (*readlink)  (struct dentry *, char *,int) ;
        int  (*follow_link)  (struct dentry *, struct nameidata *) ;
        void  (*truncate)  (struct inode *) ;
       int  (*permission)  (struct inode *, int) ;
```

```
        int  (*revalidate)  (struct dentry *) ;
        int  (*setattr)  (struct dentry *, struct iattr *) ;
        int  (*getattr)  (struct dentry *, struct iattr *) ;
};
```

表 8.4 所示为对索引节点的每个方法给予描述。

表 8.4　　　　　　　　　　索引节点对象的方法及其描述

函数形式	描述
Create(dir, dentry, mode)	在某个目录下，为与 dentry 目录项相关的常规文件创建一个新的磁盘索引节点
Lookup(dir, dentry)	查找索引节点所在的目录，这个索引节点所对应的文件名就包含在 dentry 目录项中
Link(old_dentry, dir, new_dentry)	创建一个新的名为 new_dentry 硬链接，这个新的硬连接指向 dir 目录下名为 old_dentry 的文件
unlink(dir, dentry)	从 dir 目录删除 dentry 目录项所指文件的硬链接
symlink(dir, dentry, symname)	在某个目录下，为与目录项相关的符号链创建一个新的索引节点
mkdir(dir, dentry, mode)	在某个目录下，为与目录项对应的目录创建一个新的索引节点
mknod(dir, dentry, mode, rdev)	在 dir 目录下，为与目录项对象相关的特殊文件创建一个新的磁盘索引节点。其中参数 mode 和 rdev 分别表示文件的类型和该设备的主码
rename(old_dir, old_dentry, new_dir, new_dentry)	将 old_dir 目录下的文件 old_dentry 移到 new_dir 目录下,新文件名包含在 new_dentry 指向的目录项中
readlink(dentry, buffer, buflen)	将 dentry 所指定的符号链中对应的文件路径名拷贝到 buffer 所指定的内存区
follow_link(inode, dir)	解释 inode 索引节点所指定的符号链；如果该符号链是相对路径名，从指定的 dir 目录开始进行查找
truncate(inode)	修改索引节点 inode 所指文件的长度。在调用该方法之前，必须将 inode 对象的 i_size 域设置为需要的新长度值
permission(inode, mask)	确认是否允许对 inode 索引节点所指的文件进行指定模式的访问
revalidate(dentry)	更新由目录项所指定文件的已缓存的属性（通常由网络文件系统调用）
Setattr (dentry, attr)	设置目录项的属性
Getattr (dentry, attr)	获得目录项的属性

以上这些方法均适用于所有的文件系统，但对某一个具体文件系统来说，可能只用到其中的一部分方法。例如，msdos 文件系统其公用索引节点的操作在 fs/msdos/namei.c 中定义如下：

```
    struct inode_operations msdos_dir_inode_operations = {
         create:         msdos_create,
         lookup:         msdos_lookup,
         unlink:         msdos_unlink,
         mkdir:          msdos_mkdir,
```

```
            rmdir:          msdos_rmdir,
            rename:         msdos_rename,
            setattr:        fat_notify_change,
    };
```

3．目录项操作

目录项操作是由 dentry_operations 数据结构来描述的，定义于 include/linux/dcache.h 中：

```
struct dentry_operations {
        int  (*d_revalidate)  (struct dentry *, int) ;
        int  (*d_hash)  (struct dentry *, struct qstr *) ;
        int  (*d_compare)  (struct dentry *, struct qstr *, struct qstr *) ;
        int  (*d_delete)  (struct dentry *) ;
        void  (*d_release)  (struct dentry *) ;
        void  (*d_iput)  (struct dentry *, struct inode *) ;
};
```

表 8.5 给出目录项对象的方法及其描述。

表 8.5　　　　　　　　　　　　目录项对象的方法及其描述

函数形成	描述
d_revalidate(dentry)	判定目录项是否有效。默认情况下，VFS 函数什么也不做，而网络文件系统可以指定自己的函数
d_hash(dentry, hash)	生成一个哈希值。对目录项哈希表而言，这是一个具体文件系统的哈希函数。参数 dentry 标识包含该路径分量的目录。参数 hash 指向一个结构，该结构包含要查找的路径名分量以及由 hash 函数生成的哈希值
d_compare(dir, name1, name2)	比较两个文件名。name1 应该属于 dir 所指目录。默认情况下，VFS 的这个函数就是常用的字符串匹配函数。不过，每个文件系统可用自己的方式实现这一方法。例如，MS-DOS 文件系统不区分大写和小写字母
d_delete(dentry)	如果对目录项的最后一个引用被删除（d_count 变为“0”），就调用该方法。默认情况下，VFS 的这个函数什么也不做
d_release(dentry)	当要释放一个目录项时（放入 slab 分配器），就调用该方法。默认情况下，VFS 的这个函数什么也不做
d_iput(dentry, ino)	当要丢弃目录项对应的索引节点时，就调用该方法。默认情况下，VFS 的这个函数调用 iput()释放索引节点

4．文件操作

文件操作是由 file_operations 结构来描述的，定义在 fs.h 中：

```
  /*
   * NOTE:
   * read, write, poll, fsync, readv, writev can be called
   *   without the big kernel lock held in all filesystems.
  */
 struct file_operations {
      struct module *owner;
      loff_t  (*llseek)  (struct file *, loff_t, int) ;
      ssize_t  (*read)  (struct file *, char *, size_t, loff_t *) ;
```

```
        ssize_t  (*write)  (struct file *, const char *, size_t, loff_t *) ;
        int  (*readdir)  (struct file *, void *, filldir_t) ;
        unsigned int  (*poll)  (struct file *, struct poll_table_struct *) ;
        int  (*ioctl)  (struct inode *, struct file *, unsigned int, unsigned long) ;
        int  (*mmap)  (struct file *, struct vm_area_struct *) ;
        int  (*open)  (struct inode *, struct file *) ;
        int  (*flush)  (struct file *) ;
        int  (*release)  (struct inode *, struct file *) ;
        int  (*fsync)  (struct file *, struct dentry *, int datasync) ;
        int  (*fasync)  (int, struct file *, int) ;
        int  (*lock)  (struct file *, int, struct file_lock *) ;
      ssize_t  (*readv)  (struct file *, const struct iovec *, unsigned long, loff_t *) ;
      ssize_t  (*writev)  (struct file *, const struct iovec *, unsigned long, loff_t *) ;
      ssize_t  (*sendpage)  (struct file *, struct page *, int, size_t, loff_t *, int) ;
      unsigned long  (*get_unmapped_area)  (struct file *, unsigned long, unsigned long,
unsigned long, unsigned long) ;
    };
```

这个数据结构就是连接 VFS 文件操作与具体文件系统的文件操作之间的枢纽，也是编写设备驱动程序的重要接口，后面还会给出进一步的说明。对每个函数的描述如表 8.6 所示。

表 8.6　　文件操作的描述

函数形式	描述
Owner ()	指向模块的指针。只有驱动程序才把这个域置为 THIS_MODULE，文件系统一般忽略这个域
llseek(file, offset, whence)	修改文件指针
read(file, buf, count, offset)	从文件的 offset 处开始读出 count 个字节，然后增加*offset 的值
write(file, buf, count, offset)	从文件的*offset 处开始写入 count 个字节，然后增加*offset 的值
readdir(dir, dirent, filldir)	返回 dir 所指目录的下一个目录项，这个值存入参数 dirent；参数 filldir 存放一个辅助函数的地址，该函数可以提取目录项的各个域
poll(file, poll_table)	检查是否存在关于某文件的操作事件，如果没有则睡眠，直到发生该类操作事件为止
ioctl(inode, file, cmd, arg)	向一个基本硬件设备发送命令。该方法只适用于设备文件
mmap(file, vma)	执行文件的内存映射，并将这个映射放入进程的地址空间
open(inode, file)	通过创建一个新的文件而打开一个文件，并把它链接到相应的索引节点
flush(file)	当关闭对一个打开文件的引用时，就调用该方法。也就是说，减少该文件对象 f_count 域的值。该方法的实际用途依赖于具体文件系统
release(inode, file)	释放文件对象。当关闭对打开文件的最后一个引用时，也就是说，该文件对象 f_count 域的值变为 0 时，调用该方法
fsync(file, dentry)	将 file 文件在高速缓存中的全部数据写入磁盘
fasync(file, on)	通过信号来启用或禁用异步 I/O 通告

续表

函数形式	描述
check_media_change(dev)	检测自上次对设备文件操作以来是否存在介质的改变（可以对块设备使用这一方法，因为它支持可移动介质——比如软盘和 CD-ROM）
revalidate(dev)	恢复设备的一致性（由网络文件系统使用，这是在确认某个远程设备上的介质已被改变之后才使用）
lock(file, cmd, file_lock)	对 file 文件申请一个锁
readv(file, iovec, count, offset)	与 read()类似，所不同的是，readv () 把读入的数据放在多个缓冲区中(叫缓冲区向量)
Writev(file, buf, iovec, offset)	与 write()类似。所不同的是，writev()把数据写入多个缓冲区中（叫缓冲区向量）

VFS 中定义的这个 file_operations 数据结构相当于一个标准模板，对于一个具体的文件系统来说，可能只用到其中的一些函数。注意，2.2 和 2.4 版在对 file_operations 进行初始化时有所不同，在 2.2 版中，如果某个函数没有定义，则将其置为 NULL，如：

```
struct file_operations device_fops = {
    NULL,                   /* seek */
    device_read,        /* read */
    device_write,       /* write */
    NULL,                   /* readdir */
    NULL,                   /* poll */
    NULL,                   /* ioctl */
    NULL,                   /* mmap */
    device_open,        /* open */
    NULL,                   /* flush */
    device_release      /* release */
};
```

这是标准 C 的用法，在 2.4 版中，采用了 gcc 的扩展用法，如：

```
  struct file_operations device_fops = {
    read : device_read,              /* read */
    write : device_write,        /* write */
    open : device_open,              /* open */
    release : device_release     /* release */
};
```

这种方式显然简单明了，在设备驱动程序的开发中，经常会用到这种形式。

8.3　高速缓存

8.3.1　块高速缓存

Linux 支持的文件系统大多以块的形式组织文件，为了减少对物理块设备的访问，在文件以块的形式调入内存后，使用块高速缓存（buffer_cache）对它们进行管理。每个缓冲区由两部分组成，第 1 部分称为缓冲区首部，用数据结构 buffer_head 表示，第 2 部分是真正

的缓冲区内容（即所存储的数据）。由于缓冲区首部不与数据区域相连，数据区域独立存储。因而在缓冲区首部中，有一个指向数据的指针和一个缓冲区长度的字段。图 8.6 给出了一个缓冲区的格式。

缓冲区首部包含如下内容。

- 用于描述缓冲区内容的信息，包括：所在设备号、起始物理块号、包含在缓冲区中的字节数。
- 缓冲区状态的域：是否有有用数据、是否正在使用、重新利用之前是否要写回磁盘等。
- 用于管理的域。

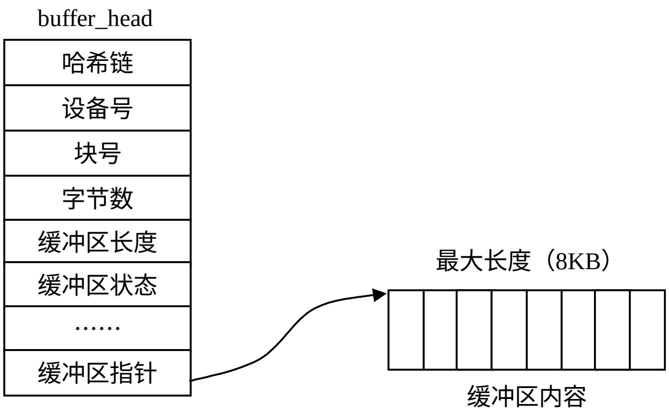


图 8.6　缓冲区格式

buffer-head 数据结构在 include\linux\fs.h 中定义如下：

```
/*
  * Try to keep the most commonly used fields in single cache lines  (16
  * bytes)  to improve performance.  This ordering should be
  * particularly beneficial on 32-bit processors.
  *
  * We use the first 16 bytes for the data which is used in searches
  * over the block hash lists  (ie. getblk ()  and friends) .
  *
  * The second 16 bytes we use for lru buffer scans, as used by
  * sync_buffers ()  and refill_freelist () .  -- sct
  */
struct buffer_head {
        /* First cache line: */
         struct buffer_head *b_next;     /* 哈希队列链表*/
         unsigned long b_blocknr;        /* 逻辑块号 */
         unsigned short b_size;          /* 块大小 */
         unsigned short b_list;          /* 本缓冲区所出现的链表 */
         kdev_t b_dev;                     /* 虚拟设备标示符 (B_FREE = free)  */

         atomic_t b_count;               /* 块引用计数器 */
          kdev_t b_rdev;                  /* 实际设备标识符*/
         unsigned long b_state;          /* 缓冲区状态位图 */
unsigned long b_flushtime;      /* 对脏缓冲区进行刷新的时间*/
```

```
        struct buffer_head *b_next_free;/* 指向 lru/free 链表中的下一个元素 */
        struct buffer_head *b_prev_free;/* 指向链表中的上一个元素*/
struct buffer_head *b_this_page;/* 每个页面中的缓冲区链表*/
        struct buffer_head *b_reqnext;  /*请求队列 */

        struct buffer_head **b_pprev;   /* 哈希队列的双向链表 */
        char * b_data;                  /* 指向数据块 */
        struct page *b_page;            /* 这个 bh 所映射的页面*/
void (*b_end_io) (struct buffer_head *bh, int uptodate); /* I/O 结束方法*/
        void *b_private;                /* 给 b_end_io 保留 */

        unsigned long b_rsector;        /* 缓冲区在磁盘上的实际位置*/
        wait_queue_head_t b_wait;     /* 缓冲区等待队列 */

        struct inode *       b_inode;
struct list_head     b_inode_buffers;   /* inode 脏缓冲区的循环链表*/
};
```

其中缓冲区状态在 fs.h 中定义为枚举类型：

```
/* bh state bits */
enum bh_state_bits {
        BH_Uptodate,    /* 如果缓冲区包含有效数据则置 1 */
        BH_Dirty,       /* 如果缓冲区数据被改变则置 1 */
        BH_Lock,        /* 如果缓冲区被锁定则置 1*/
        BH_Req,         /* 如果缓冲区数据无效则置 0  */
        BH_Mapped,      /* 如果缓冲区有一个磁盘映射则置 1 */
        BH_New,         /* 如果缓冲区为新且还没有被写出则置 1 */
        BH_Async,       /* 如果缓冲区是进行 end_buffer_io_async I/O 同步则置 1 */
        BH_Wait_IO,     /* 如果我们应该把这个缓冲区写出则置 1 */
        BH_launder,     /* 如果我们应该“清洗”这个缓冲区则置 1 */
        BH_JBD,         /* 如果与 journal_head 相连接则置 1 */

        BH_PrivateStart,/* 这不是一个状态位，但是，第 1 位由其他实体用于私有分配*/
}
```

显然一个缓冲区可以同时具有上述状态的几种。

块高速缓存的管理很复杂，下面先对空缓冲区、空闲缓冲区、正使用的缓冲区、缓冲区的大小以及缓冲区的类型作一个简短的介绍。

缓冲区可以分为两种，一种是包含了有效数据的，另一种是没有被使用的，即空缓冲区。

具有有效数据并不能表明某个缓冲区正在被使用，毕竟，在同一时间内，被进程访问的缓冲区（即处于使用状态）只有少数几个。当前没有被进程访问的有效缓冲区和空缓冲区称为空闲缓冲区。其实，buffer_head 结构中的 b_count 就可以反映出缓冲区是否处于使用状态。如果它为 0，则缓冲区是空闲的。大于 0，则缓冲区正被进程访问。

缓冲区的大小不是固定的，当前 Linux 支持 5 种大小的缓冲区，分别是 512、1024、2048、4096、8192 字节。Linux 所支持的文件系统都使用共同的块高速缓存，在同一时刻，块高速缓存中存在着来自不同物理设备的数据块，为了支持这些不同大小的数据块，Linux 使用了几种不同大小的缓冲区。

当前的 Linux 缓冲区有 3 种类型，在 include/linux/fs.h 中有如下的定义：

```
#define BUF_CLEAN        0       /*未使用的、干净的缓冲区*/
#define BUF_LOCKED       1       /*被锁定的缓冲区，正等待写入*/
#define BUF_DIRTY        2      /*脏的缓冲区，其中有有效数据，需要写回磁盘*/
```

VFS 使用了多个链表来管理块高速缓存中的缓冲区。

首先，对于包含了有效数据的缓冲区，用一个哈希表来管理，用 hash_table 来指向这个哈希表。哈希索引值由数据块号以及其所在的设备标识号计算（散列）得到。所以在 buffer_head 这个结构中有一些用于哈希表管理的域。使用哈希表可以迅速地查找到所要寻找的数据块所在的缓冲区。

对于每一种类型的未使用的有效缓冲区，系统还使用一个 LRU（最近最少使用）双链表管理，即 lru-list 链。由于共有 3 种类型的缓冲区，所以有 3 个这样的 LRU 链表。当需要访问某个数据块时，系统采取如下算法。

首先，根据数据块号和所在设备号在块高速缓存中查找，如果找到，则将它的 b_count 域加 1，因为这个域正是反映了当前使用这个缓冲区的进程数。如果这个缓冲区同时又处于某个 LRU 链中，则将它从 LRU 链中解开。

如果数据块还没有调入缓冲区，则系统必须进行磁盘 I/O 操作，将数据块调入块高速缓存，同时将空缓冲区分配一个给它。如果块高速缓存已满（即没有空缓冲区可供分配），则从某个 LRU 链首取下一个，先看是否置了“脏”位，如已置，则将它的内容写回磁盘。然后清空内容，将它分配给新的数据块。

在缓冲区使用完了后，将它的 b_count 域减 1，如果 b_count 变为 0，则将它放在某个 LRU 链尾，表示该缓冲区已可以重新利用。

为了配合以上这些操作，以及其他一些多块高速缓存的操作，系统另外使用了几个链表，主要是：

- 对于每一种大小的空闲缓冲区，系统使用一个链表管理，即 free_list 链。
- 对于空缓冲区，系统使用一个 unused_list 链管理。

以上几种链表都在 fs/buffer.c 定义。

Linux 中，用 bdflush 守护进程完成对块高速缓存的一般管理。bdflush 守护进程是一个简单的内核线程，在系统启动时运行，它在系统中注册的进程名称为 kflushd，你可以使用 ps 命令看到此系统进程。它的一个作用是监视块高速缓存中的“脏”缓冲区，在分配或丢弃缓冲区时，将对“脏”缓冲区数目作一个统计。通常情况下，该进程处于休眠状态，当块高速缓存中“脏”缓冲区的数目达到一定的比例，默认是 60%，该进程将被唤醒。但是，如果系统急需，则在任何时刻都可以唤醒这个进程。使用 update 命令可以看到和改变这个数值。

```
# update -d
```

当有数据写入缓冲区使之变成“脏”时，所有的“脏”缓冲区被连接到一个 BUF_DIRTY_LRU 链表中，bdflush 会将适当数目的缓冲区中的数据块写到磁盘上。这个数值的缺省值为 500，可以用 update 命令改变这个值。

另一个与块高速缓存管理相关的是 update 命令，它不仅仅是一个命令，还是一个后台进程。当以超级用户的身份运行时（在系统初始化时），它将周期性调用系统服务例程将老的“脏”缓冲区中内容“冲刷”到磁盘上去。它所完成的这个工作与 bdflush 类似，不同之处在于，当一个“脏”缓冲区完成这个操作后，它将把写入到磁盘上的时间标记到 buffer_head 结构中。update 每次运行时它将在系统的所有“脏”缓冲区中查找那些“冲刷”时间已经超

过一定期限的，这些过期缓冲区都要被写回磁盘。

8.3.2　索引节点高速缓存

VFS 也用了一个高速缓存来加快对索引节点的访问，和块高速缓存不同的一点是每个缓冲区不用再分为两个部分了，因为 inode 结构中已经有了类似于块高速缓存中缓冲区首部的域。索引节点高速缓存的实现代码全部在 fs/inode.c，这部分代码并没有随着内核版本的变化做更多的修改。

1．索引节点链表

每个索引节点可能处于哈希表中，也可能同时处于下列“类型”链表的一种中：

- “in_use”有效的索引节点，即 i_count > 0 且 i_nlink > 0（参看前面的 inode 结构）
- “dirty”类似于 “in_use” ，但还“脏”;
- “unused”有效的索引节点但还没使用，即 i_count = 0。

这几个链表定义如下：

```
static LIST_HEAD (inode_in_use) ;
static LIST_HEAD (inode_unused) ;
static struct list_head *inode_hashtable;
    static LIST_HEAD (anon_hash_chain) ; /* for inodes with NULL i_sb */
```

因此，索引节点高速缓存的结构概述如下。

- 全局哈希表 inode_hashtable，其中哈希值是根据每个超级块指针的值和 32 位索引节点号而得。对没有超级块的索引节点（inode->i_sb == NULL），则将其加入到 anon_hash_chain 链表的首部。例如，net/socket.c 中 sock_alloc()函数，通过调用 fs/inode.c 中 get_empty_inode()创建的套接字是一个匿名索引节点，这个节点就加入到了 anon_hash_chain 链表。
- 正在使用的索引节点链表。全局变量 inode_in_use 指向该链表中的首元素和尾元素。函数 get_empty_inode()获得一个空节点，get_new_inode()获得一个新节点，通过这两个函数新分配的索引节点就加入到这个链表中。
- 未用索引节点链表。全局变量 inode_unused 的 next 域 和 prev 域分别指向该链表中的首元素和尾元素。
- 脏索引节点链表。由相应超级块的 s_dirty 域指向该链表中的首元素和尾元素。
- 对 inode 对象的缓存，定义如下：

```
static kmem_cache_t * inode_cachep
```

这是一个 Slab 缓存，用于分配和释放索引节点对象。

索引节点的 i_hash 域指向哈希表，i_list 指向 in_use、unused 或 dirty 某个链表。所有这些链表都受单个自旋锁 inode_lock 的保护，其定义如下：

```
/*
* A simple spinlock to protect the list manipulations.
*
```

```
* NOTE! You also have to own the lock if you change
* the i_state of an inode while it is in use..
*/
static spinlock_t inode_lock = SPIN_LOCK_UNLOCKED;
```

索引节点高速缓存的初始化是由 inode_init()实现的，而这个函数是在系统启动时由 init/main.c 中的 start_kernel()函数调用的。inode_init（）只有一个参数，表示索引节点高速缓存所使用的物理页面数。因此，索引节点高速缓存可以根据可用物理内存的大小来进行配置，例如，如果物理内存足够大，就可以创建一个大的哈希表。

索引节点状态的信息存放在数据结构 inodes_stat_t 中，在 fs/fs.h 中定义如下：

```
struct inodes_stat_t {
     int nr_inodes;
     int nr_unused;
     int dummy[5];
 };
extern struct inodes_stat_t inodes_stat
```

用户程序可以通过/proc/sys/fs/inode-nr 和 /proc/sys/fs/inode-state 获得索引节点高速缓存中索引节点总数及未用索引节点数。

2．索引节点高速缓存的工作过程

为了帮助大家理解索引节点高速缓存如何工作，我们来跟踪一下在打开 Ext2 文件系统的一个常规文件时，相应索引节点的作用。

```
fd = open（"file", O_RDONLY）;
close（fd）;
```

open()系统调用是由 fs/open.c 中的 sys_open 函数实现的，而真正的工作是由 fs/open.c 中的 filp_open()函数完成的，filp_open()函数如下：

```
    struct file *filp_open（const char * filename, int flags, int mode)
{
        int namei_flags, error;
         struct nameidata nd;

        namei_flags = flags;
         if （（namei_flags+1） & O_ACCMODE)
                namei_flags++;
         if （namei_flags & O_TRUNC)
                namei_flags |= 2;

        error = open_namei（filename, namei_flags, mode, &nd）;
         if （!error)
                return dentry_open（nd.dentry, nd.mnt, flags）;

         return ERR_PTR（error）;
}
```

其中 nameidata 结构在 fs.h 中定义如下：

```
struct nameidata {
         struct dentry *dentry;
         struct vfsmount *mnt;
         struct qstr last;
```

```
            unsigned int flags;
           int last_type;
};
```

这个数据结构是临时性的，其中，我们主要关注 dentry 和 mnt 域。dentry 结构我们已经在前面介绍过，而 vfsmount 结构记录着所属文件系统的安装信息，例如文件系统的安装点、文件系统的根节点等。

filp_open（）主要调用以下两个函数。

（1）open_namei()：填充目标文件所在目录的 dentry 结构和所在文件系统的 vfsmount 结构。在 dentry 结构中 dentry->d_inode 就指向目标文件的索引节点。这个函数比较复杂和庞大，在此为了突出主题，后面我们只介绍与主题相关的内容。

（2）dentry_open()：建立目标文件的一个“上下文”，即 file 数据结构，并让它与当前进程的 task_strrct 结构挂上钩。同时，在这个函数中，调用了具体文件系统的打开函数，即 f_op->open()。该函数返回指向新建立的 file 结构的指针。

open_namei()函数通过 path_walk()与目录项高速缓存（即目录项哈希表）打交道，而 path_walk()又调用具体文件系统的 inode_operations->lookup()方法；该方法从磁盘找到并读入当前节点的目录项，然后通过 iget(sb, ino)，根据索引节点号从磁盘读入相应索引节点并在内存建立起相应的 inode 结构，这就到了我们讨论的索引节点高速缓存。

当索引节点读入内存后，通过调用 d_add(dentry, inode)，就将 dentry 结构和 inode 结构之间的链接关系建立起来。两个数据结构之间的联系是双向的。一方面，dentry 结构中的指针 d_inode 指向 inode 结构，这是一对一的关系，因为一个目录项只对应着一个文件。反之则不然，同一个文件可以有多个不同的文件名或路径（通过系统调用 link()建立，注意与符号连接的区别，那是由 symlink()建立的），所以从 inode 结构到 dentry 结构的方向是一对多的关系。因此，inode 结构的 i_ dentry 是个队列，dentry 结构通过其队列头部 d_alias 挂入相应 inode 结构的队列中。

为了进一步说明索引节点高速缓存，我们来进一步考察 iget（）。当我们打开一个文件时，就调用了 iget（）函数，而 iget 真正调用的是 iget4(sb, ino, NULL, NULL)函数，该函数代码如下：

```
        struct inode *iget4 (struct super_block *sb, unsigned long ino, find_inode_t find_actor,
void *opaque)
{
           struct list_head * head = inode_hashtable + hash (sb,ino) ;
            struct inode * inode;

            spin_lock (&inode_lock) ;
            inode = find_inode (sb, ino, head, find_actor, opaque) ;
            if  (inode)  {
                   __iget (inode) ;
                    spin_unlock (&inode_lock) ;
                   wait_on_inode (inode) ;
                   return inode;
            }
            spin_unlock (&inode_lock) ;

            /*
```

```
             * get_new_inode () will do the right thing, re-trying the search
             * in case it had to block at any point.
            */
            return get_new_inode (sb, ino, head, find_actor, opaque) ;
    }
```

下面对以上代码给出进一步的解释。

- inode 结构中有个哈希表 inode_hashtable，首先在 inode_lock 锁的保护下，通过 find_ inode 函数在哈希表中查找目标节点的 inode 结构，由于索引节点号只有在同一设备上时才是唯一的，因此，在哈希计算时要把索引节点所在设备的 super_block 结构的地址也结合进去。如果在哈希表中找到该节点，则其引用计数（i_count）加 1；如果 i_count 在增加之前为 0，说明该节点不“脏”，则该节点当前肯定处于 inode_unused list 队列中，于是，就把该节点从这个队列删除而插入 inode_in_use 队列；最后，把 inodes_stat.nr_unused 减 1。
- 如果该节点当前被加锁，则必须等待，直到解锁，以便确保 iget4()返回一个未加锁的节点。
- 如果在哈希表中没有找到该节点，说明目标节点的 inode 结构还不在内存，因此，调用 get_new_inode()从磁盘上读入相应的索引节点并建立起一个 inode 结构，并把该结构插入到哈希表中。
- 对 get_new_inode()给出进一步的说明，该函数从 slab 缓存区中分配一个新的 inode 结构，但是这个分配操作有可能出现阻塞，于是，就应当解除保护哈希表的 inode_lock 自旋锁，以便在哈希表中再次进行搜索。如果这次在哈希表中找到这个索引节点，就通过__iget 把该节点的引用计数加 1，并撤销新分配的节点；如果在哈希表中还没有找到，就使用新分配的索引节点。因此，把该索引节点的一些域先初始化为必须的值，然后调用具体文件系统的 sb->s_op->read_inode()域填充该节点的其他域。这就把我们从索引节点高速缓存带到了某个具体文件系统的代码中。当 s_op->read_inode()方法正在从磁盘读索引节点时，该节点被加锁（i_state = I_LOCK）；当 read_inode()返回时，该节点的锁被解除，并且唤醒所有等待者。

8.3.3 目录高速缓存

由于从磁盘读入一个目录项并构造相应的目录项对象需要花费大量的时间，所以，在完成对目录项对象的操作后，可能后面还要使用它，因此在内存仍保留它有重要的意义。例如，我们经常需要编辑文件，随后进行编译或编辑，然后打印或拷贝，再进行编辑，诸如此类的情况中，同一个文件需要被反复访问。

每个目录项对象属于以下 4 种状态之一。

- 空闲状态：处于该状态的目录项对象不包含有效的信息，还没有被 VFS 使用。它对应的内存区由 slab 分配器进行管理。
- 未使用状态：处于该状态的目录项对象当前还没有被内核使用。该对象的引用计数器 d_count 的值为 NULL。但其 d_inode 域仍然指向相关的索引节点。该目录项对象包含有效的信息，但为了在必要时回收内存，它的内容可能被丢弃。

- 正在使用状态：处于该状态的目录项对象当前正在被内核使用。该对象的引用计数器 d_count 的值为正数，而其 d_inode 域指向相关的索引节点对象。该目录项对象包含有效的信息，并且不能被丢弃。
- 负状态：与目录项相关的索引节点不复存在，那是因为相应的磁盘索引节点已被删除。该目录项对象的 d_inode 域置为 NULL，但该对象仍然被保存在目录项高速缓存中，以便后续对同一文件目录名的查找操作能够快速完成，术语“负的”容易使人误解，因为根本不涉及任何负值。

为了最大限度地提高处理这些目录项对象的效率，Linux 使用目录项高速缓存，它由以下两种类型的数据结构组成。

- 处于正在使用、未使用或负状态的目录项对象的集合。
- 一个哈希表，从中能够快速获取与给定的文件名和目录名对应的目录项对象。如果访问的对象不在目录项高速缓存中，哈希函数返回一个空值。

目录项高速缓存的作用也相当于索引节点高速缓存的控制器。内核内存中，目录项可能已经不使用，但与其相关的索引节点并不被丢弃，这是由于目录项高速缓存仍在使用它们，因此，索引节点的 i_count 域不为空。于是，这些索引节点对象还保存在 RAM 中，并能够借助相应的目录项快速引用它们。

所有“未使用”目录项对象都存放在一个“最近最少使用”的双向链表中，该链表按照插入的时间排序。换句话说，最后释放的目录项对象放在链表的首部，所以最近最少使用的目录项对象总是靠近链表的尾部。一旦目录项高速缓存的空间开始变小，内核就从链表的尾部删除元素，使得多数最近经常使用的对象得以保留。LRU 链表的首元素和尾元素的地址存放在变量 dentry_unused 中的 next 域和 prev 域中。目录项对象的 d_lru 域包含的指针指向该链表中相邻目录的对象。

每个“正在使用”的目录项对象都被插入一个双向链表中，该链表由相应索引节点对象的 i_dentry 域所指向（由于每个索引节点可能与若干硬链接关联，所以需要一个链表）。目录项对象的 d_alias 域存放链表中相邻元素的地址。

当指向相应文件的最后一个硬链接被删除后，一个“正在使用”的目录项对象可能变成“负”状态。在这种情况下，该目录项对象被移到“未使用”目录项对象组成的 LRU 链表中。每当内核缩减目录项高速缓存时，“负”状态目录项对象就朝着 LRU 链表的尾部移动，这样一来，这些对象就逐渐被释放。

哈希表是由 dentry_hashtable 数组实现的。数组中的每个元素是一个指向链表的指针，这种链表就是把具有相同哈希表值的目录项进行散列而形成的。该数组的长度取决于系统已安装 RAM 的数量。目录项对象的 d_hash 域包含指向具有相同 hash 值的链表中的相邻元素。哈希函数产生的值是由目录及文件名的目录项对象的地址计算出的。

8.4 文件系统的注册、安装与卸载

8.4.1 文件系统的注册

当内核被编译时，就已经确定了可以支持哪些文件系统，这些文件系统在系统引导时，在 VFS 中进行注册。如果文件系统是作为内核可装载的模块，则在实际安装时进行注册，并在模块卸载时注销。每个文件系统都有一个初始化例程，它的作用就是在 VFS 中进行注册，即填写一个叫做 file_system_type 的数据结构，该结构包含了文件系统的名称以及一个指向对应的 VFS 超级块读取例程的地址，所有已注册的文件系统的 file_system_type 结构形成一个链表，为区别后面将要说到的已安装的文件系统形成的另一个链表，我们把这个链表称为注册链表。图 8.7 所示就是内核中的 file_system_type 链表，链表头由 file_systems 变量指定。

图 8.7 仅示意性地说明系统中已安装的 3 个文件系统 Ext2、proc、iso9660 其 file_system_type 结构所形成的链表。当然，系统中实际安装的文件系统要更多。

file_system_type 的数据结构在 fs.h 中定义如下：

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block * (*read_super) (struct super_block *, void *, int) ;
        struct module *owner;
        struct file_system_type * next;
        struct list_head fs_supers;
};
```

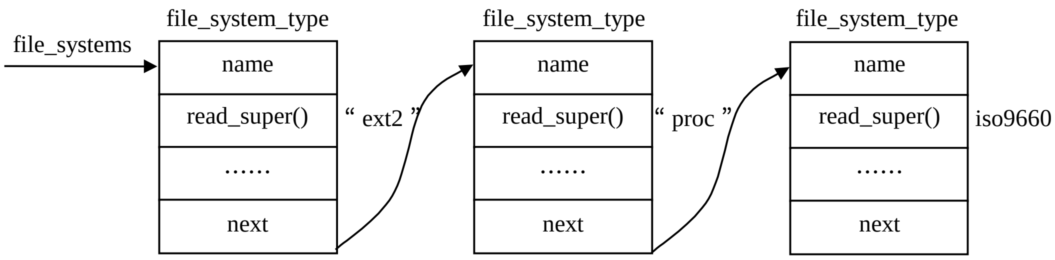


图 8.7 已注册的文件系统形成的链表

对其中几个域的说明如下。

- name：文件系统的类型名，以字符串的形式出现。
- fs_flags：指明具体文件系统的一些特性，有关标志定义于 fs.h 中：

```
    /* public flags for file_system_type */
    #define FS_REQUIRES_DEV 1
#define FS_NO_DCACHE    2 /* Only dcache the necessary things. */
    #define FS_NO_PRELIM    4 /* prevent preloading of dentries, even if
                               * FS_NO_DCACHE is not set.
                              */
```

```
#define FS_SINGLE       8 /* Filesystem that can have only one superblock */
#define FS_NOMOUNT      16 /* Never mount from userland */
#define FS_LITTER       32 /* Keeps the tree in dcache */
#define FS_ODD_RENAME   32768   /* Temporary stuff; will go away as soon
                                 * as nfs_rename ()  will be cleaned up
                                  */
```

对某些常用标志的说明如下。

（1）有些虚拟的文件系统，如 pipe、共享内存等，根本不允许由用户进程通过系统调用 mount()来安装。这样的文件系统其 fs_flags 中的 FS_NOMOUNT 标志位为 1。

（2）一般的文件系统类型要求有物理的设备作为其物质基础，其 fs_flags 中的 FS_REQUIRES_DEV 标志位为 1，这些文件系统如 Ext2、Minix、ufs 等。

（3）有些虚拟文件系统在安装了同类型中的第 1 个“设备”，从而创建了其超级块的 super_block 数据结构，在安装同一类型中的其他设备时就共享已存在的 super_block 结构，而不再有自己的超级块结构。此时 fs_flags 中的 FS_SINGLE 标志位为 1，表示整个文件系统只有一个超级块，而不像一般的文件系统类型那样，每个具体的设备上都有一个超级块。

- read_super：这是各种文件系统读入其超级块的函数指针。因为不同的文件系统其超级块不同，因此其读入函数也不同。
- owner：如果 file_system_type 所代表的文件系统是通过可安装模块实现的，则该指针指向代表着具体模块的 module 结构。如果文件系统是静态地链接到内核，则这个域为 NULL。实际上，你只需要把这个域置为 THIS_MODLUE（这是个一个宏），它就能自动地完成上述工作。
- next：把所有的 file_system_type 结构链接成单项链表的链接指针，变量 file_systems 指向这个链表。这个链表是一个临界资源，受 file_systems_lock 自旋读写锁的保护。
- fs_supers：这个域是 Linux 2.4.10 以后的内核版本中新增加的，这是一个双向链表。链表中的元素是超级块结构。如前说述，每个文件系统都有一个超级块，但有些文件系统可能被安装在不同的设备上，而且每个具体的设备都有一个超级块，这些超级块就形成一个双向链表。

搞清楚这个数据结构的各个域以后，就很容易理解下面的注册函数 register_filesystem（），该函数定义于 fs/super.c：

```
     /**
     *      register_filesystem - register a new filesystem
     *      @fs: the file system structure
*
*      Adds the file system passed to the list of file systems the kernel
*      is aware of for mount and other syscalls. Returns 0 on success,
*      or a negative errno code on an error.
*
*      The &struct file_system_type that is passed is linked into the kernel
*      structures and must not be freed until the file system has been
*      unregistered.
 */

int register_filesystem (struct file_system_type * fs)
```

```
{
        int res = 0;
        struct file_system_type ** p;

        if  (!fs)
                return -EINVAL;
        if  (fs->next)
                return -EBUSY;
        INIT_LIST_HEAD (&fs->fs_supers) ;
        write_lock (&file_systems_lock) ;
        p = find_filesystem (fs->name) ;
        if  (*p)
                res = -EBUSY;
        else
                *p = fs;
        write_unlock (&file_systems_lock) ;
        return res;
}
```

find_filesystem（）函数在同一个文件中定义如下：

```
  static struct file_system_type **find_filesystem (const char *name)
{
        struct file_system_type **p;
        for  (p=&file_systems; *p; p=& (*p) ->next)
                if  (strcmp ( (*p) ->name,name)  == 0)
                        break;
        return p;
}
```

注意，对注册链表的操作必须互斥地进行，因此，对该链表的查找加了写锁 write_lock。

文件系统注册后，还可以撤消这个注册，即从注册链表中删除一个 file_system_type 结构，此后系统不再支持该种文件系统。fs/super.c 中的 unregister_filesystem()函数就是起这个作用的，它在执行成功后返回 0，如果注册链表中本来就没有指定的要删除的结构，则返回-1，其代码如下：

```
/**
*       unregister_filesystem - unregister a file system
*       @fs: filesystem to unregister
*
*       Remove a file system that was previously successfully registered
*       with the kernel. An error is returned if the file system is not found.
*       Zero is returned on a success.
*
*       Once this function has returned the &struct file_system_type structure
*       may be freed or reused.
*/

int unregister_filesystem (struct file_system_type * fs)
{
        struct file_system_type ** tmp;

        write_lock (&file_systems_lock) ;
        tmp = &file_systems;
```

```
        while  (*tmp)  {
               if  (fs == *tmp)  {
                       *tmp = fs->next;
                       fs->next = NULL;
                        write_unlock (&file_systems_lock) ;
                       return 0;
                }
               tmp = & (*tmp) ->next;
        }
        write_unlock (&file_systems_lock) ;
        return -EINVAL;
 }
```

8.4.2　文件系统的安装

要使用一个文件系统，仅仅注册是不行的，还必须安装这个文件系统。在安装 Linux 时，硬盘上已经有一个分区安装了 Ext2 文件系统，它是作为根文件系统的，根文件系统在启动时自动安装。其实，在系统启动后你所看到的文件系统，都是在启动时安装的。如果需要自己（一般是超级用户）安装文件系统，则需要指定 3 种信息：文件系统的名称、包含文件系统的物理块设备、文件系统在已有文件系统中的安装点。例如：

```
$ mount -t iso9660 /dev/hdc /mnt/cdrom
```

其中，iso9660 就是文件系统的名称，/dev/hdc 是包含文件系统的物理块设备，/mnt/cdrom 是将要安装到的目录，即安装点。从这个例子可以看出，安装一个文件系统实际上是安装一个物理设备。

把一个文件系统（或设备）安装到一个目录点时要用到的主要数据结构为 vfsmount，定义于 include/linux/mount.h 中：

```
   struct vfsmount
{
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;    /* fs we are mounted on */
        struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
        struct dentry *mnt_root;        /* root of the mounted tree */
         struct super_block *mnt_sb;     /* pointer to superblock */
         struct list_head mnt_mounts;    /* list of children, anchored here */
         struct list_head mnt_child;     /* and going through their mnt_child */
         atomic_t mnt_count;
        int mnt_flags;
         char *mnt_devname;              /* Name of device e.g. /dev/dsk/hda1 */
         struct list_head mnt_list;
};
```

下面对结构中的主要域给予进一步说明。

- 为了对系统中的所有安装点进行快速查找，内核把它们按哈希表来组织，mnt_hash 就是形成哈希表的队列指针。
- mnt_mountpoint 是指向安装点 dentry 结构的指针。而 dentry 指针指向安装点所在目录树中根目录的 dentry 结构。

- mnt_parent 是指向上一层安装点的指针。如果当前的安装点没有上一层安装点（如根设备），则这个指针为 NULL。同时，vfsmount 结构中还有 mnt_mounts 和 mnt_child 两个队列头，只要上一层 vfsmount 结构存在，就把当前 vfsmount 结构中 mnt_child 链入上一层 vfsmount 结构的 mnt_mounts 队列中。这样就形成一个设备安装的树结构，从一个 vfsmount 结构的 mnt_mounts 队列开始，可以找到所有直接或间接安装在这个安装点上的其他设备。
- mnt_sb 指向所安装设备的超级块结构 super_blaock。
- mnt_list 是指向 vfsmount 结构所形成链表的头指针。

另外，系统还定义了 vfsmntlist 变量，指向 mnt_list 队列。对这个数据结构的进一步理解请看后面文件系统安装的具体实现过程。

文件系统的安装选项，也就是 vfsmount 结构中的安装标志 mnt_flags 在 linux/fs.h 中定义如下：

```
/*
* These are the fs-independent mount-flags: up to 32 flags are supported
*/
 #define MS_RDONLY        1      /* Mount read-only */
 #define MS_NOSUID        2      /* Ignore suid and sgid bits */
 #define MS_NODEV         4     /* Disallow access to device special files */
 #define MS_NOEXEC        8      /* Disallow program execution */
 #define MS_SYNCHRONOUS  16      /* Writes are synced at once */
 #define MS_REMOUNT      32      /* Alter flags of a mounted FS */
 #define MS_MANDLOCK     64      /* Allow mandatory locks on an FS */
 #define MS_NOATIME      1024    /* Do not update access times. */
 #define MS_NODIRATIME   2048    /* Do not update directory access times */
 #define MS_BIND         4096
 #define MS_MOVE         8192
 #define MS_REC          16384
#define MS_VERBOSE      32768
#define MS_ACTIVE        (1<<30)
#define MS_NOUSER        (1<<31)
```

从定义可以看出，每个标志对应 32 位中的一位。安装标志是针对整个文件系统中的所有文件的。例如，如果 MS_NOSUID 标志为 1，则整个文件系统中所有可执行文件的 suid 标志位都不起作用了。其他安装标志的具体含义在后面介绍 do_mount()函数代码时再进一步介绍。

1．安装根文件系统

每个文件系统都有它自己的根目录，如果某个文件系统（如 Ext2）的根目录是系统目录树的根目录，那么该文件系统称为根文件系统。而其他文件系统可以安装在系统的目录树上，把这些文件系统要插入的那些目录就称为安装点。

当系统启动时，就要在变量 ROOT_DEV 中寻找包含根文件系统的磁盘主码。当编译内核或向最初的启动装入程序传递一个合适的选项时，根文件系统可以被指定为/dev 目录下的一个设备文件。类似地，根文件系统的安装标志存放在 root_mountflags 变量中。用户可以指定这些标志，这是通过对已编译的内核映像执行/sbin/rdev 外部程序，或者向最初的启动装入程序传递一个合适的选项来达到的。根文件系统的安装函数为 mount_root()。

2．安装一个常规文件系统

一旦在系统中安装了根文件系统，就可以安装其他的文件系统。每个文件系统都可以安装在系统目录树中的一个目录上。

前面我们介绍了以命令方式来安装文件系统，在用户程序中要安装一个文件系统则可以调用 mount（）系统调用。mount()系统调用在内核的实现函数为 sys_mount()，其代码在 fs/namespace.c 中。

```
asmlinkage long sys_mount(char * dev_name, char * dir_name, char * type,
                          unsigned long flags, void * data)
{
        int retval;
        unsigned long data_page;
        unsigned long type_page;
        unsigned long dev_page;
        char *dir_page;

        retval = copy_mount_options （type, &type_page）;
        if （retval < 0)
               return retval;

        dir_page = getname （dir_name）;
        retval = PTR_ERR （dir_page）;
        if （IS_ERR （dir_page））
               goto out1;

        retval = copy_mount_options （dev_name, &dev_page）;
        if （retval < 0)
               goto out2;

        retval = copy_mount_options （data, &data_page）;
        if （retval < 0)
               goto out3;

        lock_kernel （）;
        retval = do_mount （（char*）dev_page, dir_page, （char*）type_page,
                          flags, （void*）data_page）;
        unlock_kernel （）;
        free_page （data_page）;

out3:
        free_page （dev_page）;
out2:
        putname （dir_page）;
out1:
        free_page （type_page）;
        return retval;
}
```

下面给出进一步的解释。

- 参数 dev_name 为待安装文件系统所在设备的路径名，如果不需要就为空（例如，当

待安装的是基于网络的文件系统时）；dir_name 则是安装点（空闲目录）的路径名；type 是文件系统的类型，必须是已注册文件系统的字符串名（如“Ext2”，“MSDOS”等）；flags 是安装模式，如前面所述。data 指向一个与文件系统相关的数据结构（可以为 NULL）。

- copy_mount_options（）和 getname()函数将结构形式或字符串形式的参数值从用户空间拷贝到内核空间。这些参数值的长度均以一个页面为限，但是 getname()在复制时遇到字符串结尾符“\0”就停止，并返回指向该字符串的指针；而 copy_mount_options（）则拷贝整个页面，并返回该页面的起始地址。

该函数调用的主要函数为 do_mount()，do_mount()执行期间要加内核锁，不过这个锁是针对 SMP，我们暂不考虑。do_mount()的实现代码在 fs/namespace.c 中：

```
long do_mount (char * dev_name, char * dir_name, char *type_page,
                   unsigned long flags, void *data_page)
{
        struct nameidata nd;
        int retval = 0;
        int mnt_flags = 0;

        /* Discard magic */
         if  ( (flags & MS_MGC_MSK)  == MS_MGC_VAL)
                flags &= ~MS_MGC_MSK;

         /* Basic sanity checks */

         if  (!dir_name || !*dir_name || !memchr (dir_name, 0, PAGE_SIZE) )
                 return -EINVAL;
         if  (dev_name && !memchr (dev_name, 0, PAGE_SIZE) )
                 return -EINVAL;

        /* Separate the per-mountpoint flags */
         if  (flags & MS_NOSUID)
                 mnt_flags |= MNT_NOSUID;
         if  (flags & MS_NODEV)
                 mnt_flags |= MNT_NODEV;
         if  (flags & MS_NOEXEC)
                mnt_flags |= MNT_NOEXEC;
        flags &= ~ (MS_NOSUID|MS_NOEXEC|MS_NODEV) ;

        /* ... and get the mountpoint */
         if  (path_init (dir_name, LOOKUP_FOLLOW|LOOKUP_POSITIVE, &nd) )
                retval = path_walk (dir_name, &nd) ;
         if  (retval)
                 return retval;

        if  (flags & MS_REMOUNT)
                retval = do_remount (&nd, flags & ~MS_REMOUNT, mnt_flags,
                                     data_page) ;
         else if  (flags & MS_BIND)
                 retval = do_loopback (&nd, dev_name, flags & MS_REC) ;
         else if  (flags & MS_MOVE)
                 retval = do_move_mount (&nd, dev_name) ;
```

```
        else
                retval = do_add_mount (&nd, type_page, flags, mnt_flags,
                                       dev_name, data_page) ;
        path_release (&nd) ;
        return retval;
}
```

下面对函数中的主要代码给予解释。

• MS_MGC_VAL 和 MS_MGC_MSK 是在以前的版本中定义的安装标志和掩码，现在的安装标志中已经不使用这些魔数了，因此，当还有这个魔数时，则丢弃它。

• 对参数 dir_name 和 dev_name 进行基本检查，注意“ !dir_name ” 和“ !*dir_name ”的不同，前者指指向字符串的指针为不为空，而后者指字符串不为空。memchr()函数在指定长度的字符串中寻找指定的字符，如果字符串中没有结尾符“ \0 ”，也是一种错误。前面已说过，对于基于网络的文件系统 dev_name 可以为空。

• 把安装标志为 MS_NOSUID、MS_NOEXEC 和 MS_NODEV 的 3 个标志位从 flags 分离出来，放在局部安装标志变量 mnt_flags 中。

• 函数 path_init () 和 path_walk () 寻找安装点的 dentry 数据结构，找到的 dentry 结构存放在局部变量 nd 的 dentry 域中。

• 如果 flags 中的 MS_REMOUNT 标志位为 1，就表示所要求的只是改变一个原已安装设备的安装方式，例如从“ 只读“ 安装方式改为“ 可写 ”安装方式，这是通过调用 do_remount () 函数完成的。

• 如果 flags 中的 MS_BIND 标志位为 1，就表示把一个“ 回接 ”设备捆绑到另一个对象上。回接设备是一种特殊的设备（虚拟设备），而实际上并不是一种真正设备，而是一种机制，这种机制提供了把回接设备回接到某个可访问的常规文件或块设备的手段。通常在/dev 目录中有/dev/loop0 和/dev/loop1 两个回接设备文件。调用 do_loopback () 来实现回接设备的安装。

• 如果 flags 中的 MS_MOVE 标志位为 1，就表示把一个已安装的设备可以移到另一个安装点，这是通过调用 do_move_mount () 函数来实现的。

• 如果不是以上 3 种情况，那就是一般的安装请求，于是把安装点加入到目录树中，这是通过调用 do_add_mount () 函数来实现的，而 do_add_mount () 首先调用 do_kern_mount () 函数形成一个安装点，该函数的代码在 fs/super.c 中：

```
    struct vfsmount *do_kern_mount (char *type, int flags, char *name, void *data)
{
        struct file_system_type * fstype;
        struct vfsmount *mnt = NULL;
        struct super_block *sb;

        if  (!type || !memchr (type, 0, PAGE_SIZE) )
                return ERR_PTR (-EINVAL) ;

        /* we need capabilities... */
        if  (!capable (CAP_SYS_ADMIN) )
                return ERR_PTR (-EPERM) ;

        /* ... filesystem driver... */
```

```
        fstype = get_fs_type (type) ;
        if (!fstype)
                return ERR_PTR (-ENODEV) ;

        /* ... allocated vfsmount... */
        mnt = alloc_vfsmnt () ;
        if (!mnt) {
                mnt = ERR_PTR (-ENOMEM) ;
                goto fs_out;
        }
        set_devname (mnt, name) ;
        /* get locked superblock */
        if (fstype->fs_flags & FS_REQUIRES_DEV)
                sb = get_sb_bdev (fstype, name, flags, data) ;
        else if (fstype->fs_flags & FS_SINGLE)
                sb = get_sb_single (fstype, flags, data) ;
        else
                sb = get_sb_nodev (fstype, flags, data) ;

        if (IS_ERR (sb) ) {
                free_vfsmnt (mnt) ;
                mnt = (struct vfsmount *) sb;
                goto fs_out;
        }
        if (fstype->fs_flags & FS_NOMOUNT)
                sb->s_flags |= MS_NOUSER;

        mnt->mnt_sb = sb;
        mnt->mnt_root = dget (sb->s_root) ;
        mnt->mnt_mountpoint = mnt->mnt_root;
        mnt->mnt_parent = mnt;
        up_write (&sb->s_umount) ;
fs_out:
        put_filesystem (fstype) ;
        return mnt;
}
```

对该函数的解释如下。

- 只有系统管理员才具有安装一个设备的权力，因此首先要检查当前进程是否具有这种权限。
- get_fs_type () 函数根据具体文件系统的类型名在 file_system_file 链表中找到相应的结构。
- alloc_vfsmnt () 函数调用 slab 分配器给类型为 vfsmount 结构的局部变量 mnt 分配空间，并进行相应的初始化。
- set_devname () 函数设置设备名。
- 一般的文件系统类型要求有物理的设备作为其物质基础，如果 fs_flags 中的 FS_REQUIRES_DEV 标志位为 1，说明这就是正常的文件系统类型，如 Ext2、mnix 等。对于这种文件系统类型，通过调用 get_sb_bdev () 从待安装设备上读其超级块。
- 如果 fs_flags 中的 FS_SINGLE 标志位为 1，说明整个文件系统只有一个类型，也就

是说，这是一种虚拟的文件系统类型。这种文件类型在安装了同类型的第 1 个“设备”，通过调用 get_sb_single（）创建了超级块 super_block 结构后，再安装的同类型设备就共享这个数据结构。但是像 Ext2 这样的文件系统类型在每个具体设备上都有一个超级块。

• 还有些文件系统类型的 fs_flags 中的 FS_NOMOUNT、FS_REUIRE_DEV 以及 FS_SINGLE 标志位全都为 0，那么这些所谓的文件系统其实是“虚拟的”，通常只是用来实现某种机制或者规程，所以根本就没有对应的物理设备。对于这样的文件系统类型都是通过 get_sb_nodev（）来生成一个 super_block 结构的。

• 如果文件类型 fs_flags 的 FS_NOMOUNT 标志位为 1，说明根本就没有用户进行安装，因此，把超级块中的 MS_NOUSER 标志位置 1。

• mnt->mnt_sb 指向所安装设备的超级块 sb；mnt->mnt_root 指向其超级块的根 b->s_root，dget（）函数把 dentry 的引用计数 count 加 1；mnt->mnt_mountpoint 也指向超级块的根，而 mnt->mnt_parent 指向自己。到此为止，仅仅形成了一个安装点，但还没有把这个安装点挂接在目录树上。

下面我们来看 do_add_mount（）的代码：

```
   static int do_add_mount (struct nameidata *nd, char *type, int flags,
                       int mnt_flags, char *name, void *data)
{
         struct vfsmount *mnt = do_kern_mount (type, flags, name, data) ;
         int err = PTR_ERR (mnt) ;

         if  (IS_ERR (mnt) )
                goto out;

         down (&mount_sem) ;
         /* Something was mounted here while we slept */
         while (d_mountpoint (nd->dentry)  && follow_down (&nd->mnt, &nd->dentry) )
                ;
         err = -EINVAL;
         if  (!check_mnt (nd->mnt) )
                goto unlock;

        /* Refuse the same filesystem on the same mount point */
        err = -EBUSY;
         if  (nd->mnt->mnt_sb == mnt->mnt_sb && nd->mnt->mnt_root == nd->dentry)
                 goto unlock;

        mnt->mnt_flags = mnt_flags;
        err = graft_tree (mnt, nd) ;
unlock:
         up (&mount_sem) ;
         mntput (mnt) ;
 out:
         return err;
   }
```

下面是对以上代码的解释。

• 首先检查 do_kern_mount（）所形成的安装点是否有效。

• 在 do_mount()函数中，path_init()和 path_walk()函数已经找到了安装点的 dentry

结构、inode 结构以及 vfsmount 结构，并存放在类型为 nameidata 的局部变量 nd 中，在 do_add_mount()中通过参数传递了过来。

- 但是，在 do_kern_mount () 函数中从设备上读入超级块的过程是个较为漫长的过程，当前进程在等待从设备上读入超级块的过程中几乎可肯定要睡眠，这样就有可能另一个进程捷足先登抢先将另一个设备安装到了同一个安装点上。d_mountpoint () 函数就是检查是否发生了这种情况。如果确实发生了这种情况，其对策就是调用 follow_down () 前进到已安装设备的根节点，并且通过 while 循环进一步检测新的安装点，直到找到一个空安装点为止。
- 如果在同一个安装点上要安装两个同样的文件系统，则出错。
- 调用 graft_tree () 把 mnt 与安装树挂接起来，完成最终的安装。
- 至此，设备的安装就完成了。

8.4.3 文件系统的卸载

如果文件系统中的文件当前正在使用，该文件系统是不能被卸载的。如果文件系统中的文件或目录正在使用，则 VFS 索引节点高速缓存中可能包含相应的 VFS 索引节点。根据文件系统所在设备的标识符，检查在索引节点高速缓存中是否有来自该文件系统的 VFS 索引节点，如果有且使用计数大于 0，则说明该文件系统正在被使用，因此，该文件系统不能被卸载。否则，查看对应的 VFS 超级块，如果该文件系统的 VFS 超级块标志为“脏”，则必须将超级块信息写回磁盘。上述过程结束之后，对应的 VFS 超级块被释放，vfsmount 数据结构将从 vfsmntlist 链表中断开并被释放。具体的实现代码为 fs/super.c 中的 sysy_umount () 函数，在此不再进行详细的讨论。

8.5 限额机制

设想一下，如果对用户不采取某些限制措施，则任一用户可能用完文件系统的所有可用空间，在某些环境中，这种情况是不能接受的。Linux 中为了限制一个用户可获得的文件资源数量，使用了限额机制。限额机制对一个用户可分配的文件数目和可使用的磁盘空间设置了限制。系统管理员能分别为每一用户设置限额。

限制有软限制和硬限制之分，硬限制是绝对不允许超过的，而软限制则由系统管理员来确定。当用户占用的资源超过软限制时，系统开始启动定时机制，并在用户的终端上显示警告信息，但并不终止用户进程的运行，如果在规定时间内用户没有采取积极措施改正这一问题，则软限制将被强迫转化为硬限制，用户的进程将被中止。这个规定的时间可以由系统管理员来设置，默认为一周，在 include/linux/quota.h 中有如下宏定义：

```
#define MAX_IQ_TIME          604800 /*  (7*24*60*60)  =1 周 */
#define MAX_DQ_TIME          604800  /*  (7*24*60*60)  =1 周 */
```

分别是超过索引节点软限制的最长允许时间和超过块的软限制的最长允许时间。

下面看一下在 Linux 中，限额机制具体是怎样实现的。

首先，在编译内核时，要选择“支持限额机制”一项，默认情况下，Linux 不使用限额

机制。如果使用了限额机制，每一个安装的文件系统都与一个限额文件相联系，限额文件通常驻留在文件系统的根目录里，它实际是一组以用户标识号来索引的限额记录，每个限额记录可称为一个限额块，其数据结构如下（在 inclue/linux/quota.h 中定义）：

```
struct dqblk {
    __u32 dqb_bhardlimit;   /* 块的硬限制*/
    __u32 dqb_bsoftlimit;     /* 块的软限制 */
    __u32 dqb_curblocks;      /* 当前占有的块数 */
    __u32 dqb_ihardlimit;    /* 索引节点的硬限制 */
    __u32 dqb_isoftlimit;    /* 索引节点的软限制 */
    __u32 dqb_curinodes;     /* 当前占用的索引节点数 */
    time_t dqb_btime;       /* 块的软限制变为硬限制前，剩余的警告次数*/
    time_t dqb_itime;      /* 索引节点的软限制变为硬限制前，剩余的警告次数 */
};
```

限额块调入内存后，用哈希表来管理，这就要用到另一个结构 dquot（也在 inclue/linux/quota.h 中定义），其数据结构如下：

```
   struct dquot {
         struct list_head dq_hash;       /*在内存的哈希表*/
         struct list_head dq_inuse;      /*正在使用的限额块组成的链表*/
         struct list_head dq_free;       /* 空闲限额块组成的链表 */
         wait_queue_head_t dq_wait_lock; /* 指向加锁限额块的等待队列*/
         wait_queue_head_t dq_wait_free; /* 指向未用限额块的等待队列*/
         int dq_count;                   /* 引用计数 */

         /* fields after this point are cleared when invalidating */
         struct super_block *dq_sb;      /* superblock this applies to */
         unsigned int dq_id;             /* ID this applies to (uid, gid)  */
         kdev_t dq_dev;                  /* Device this applies to */
         short dq_type;                  /* Type of quota */
         short dq_flags;                 /* See DQ_* */
         unsigned long dq_referenced;    /* Number of times this dquot was
                                            referenced during its lifetime */
         struct dqblk dq_dqb;            /* Diskquota usage */
};
```

哈希表是用文件系统所在的设备号和用户标识号为散列关键值的。

vfs 的索引节点结构中有一个指向 dquot 结构的指针。也就是说，调入内存的索引节点都要与相应的 dquot 结构联系，dquot 结构中，引用计数就是反映了当前有几个索引节点与之联系，只有在引用计数为 0 时，才将该结构放入空闲链表中。

如果使用了限额机制，则当有新的块分配请求，系统要以文件拥有者的标识号为索引去查找限额文件中相应的限额块，如果限额并没有满，则接受请求，并把它加入使用计数中。如果已达到或超过限额，则拒绝请求，并返回错误信息。

下面是为一个用户设置限额的具体实现方法。

（1）检查/etc/fstab ，如果没有提供限额机制，则该文件类似下面这样。

```
/dev/hda1 / Ext2 defaults 1 1
/dev/hda2 /home Ext2 defaults 1 2
```

（2）为了设置用户 user1 在目录 /home/user1 下所占用磁盘空间和使用文件数的限额，将/etc/fstab 改成像下面这样。

```
/dev/hda1 / Ext2 defaults 1 1
/dev/hda2 /home Ext2 defaults,usrquota 1 2
```

（3）以 root 登录，在需要设置限额的分区目录下创建空文件 quota.user。

```
#touch /home/quota.user
#chmod 600 /home/quota.user
```

（4）重新启动机器。

（5）为指定的用户分配磁盘空间和最多存放文件个数。

```
# edquota -u user1
Quota for user user1
/dev/hda2: blocks in use:10, limits （soft=4000,hard=5400）
inodes in use : 400, limits （soft=1200,hard=1600）
```

你只需对 limits 那一项进行修改即可。

用#quota user1 可以查看用户 user1 的磁盘限额设置情况。

8.6 具体文件系统举例

如前所述，每种文件系统类型都有个 file_system_type 结构，而结构中的 fs_flags 则由各种标志位组成，这些标志位表明了具体文件系统类型的特性，也决定着这种文件系统的安装过程。以物理设备为基础的常规文件系统类型（如 Ext2、Minix 等），由用户进程通过系统调用 mount()来安装，而有些没有物理设备对应的文件系统（如 pipe、共享内存区等），由内核通过 kern_mount()来安装。

内核代码中提供了两个用来建立 file_system_type 结构的宏，其定义在 fs.h 中：

```
   #define DECLARE_FSTYPE（var,type,read,flags） \
struct file_system_type var = { \
         name:           type, \
         read_super:     read, \
         fs_flags:       flags, \
        owner:          THIS_MODULE, \
}

 #define DECLARE_FSTYPE_DEV（var,type,read） \
         DECLARE_FSTYPE（var,type,read,FS_REQUIRES_DEV）
```

一般常规的文件系统类型都通过 DECLARE_FSTYPE_DEV 建立其结构，因为它们的 FS_REQUIRES_DEV 标志位为 1，而其他标志位为 0。相比之下，特殊的、虚拟的文件系统类型大多直接通过 DECLARE_FSTYPE 建立其结构，因为它们的 fs_flags 是特殊的。

8.6.1 管道文件系统 pipefs

pipefs 是一种简单的、虚拟的文件系统类型，因为它没有对应的物理设备，因此其安装时不需要块设备，在第十章将看到，大部分文件系统是以模块的形式来实现的。该文件系统相关的代码在 fs/pipe.c 中：

```
static DECLARE_FSTYPE（pipe_fs_type, "pipefs", pipefs_read_super,
              FS_NOMOUNT|FS_SINGLE）;
```

```
static int __init init_pipe_fs (void)
{
        int err = register_filesystem (&pipe_fs_type) ;
        if  (!err)  {
                pipe_mnt = kern_mount (&pipe_fs_type) ;
                err = PTR_ERR (pipe_mnt) ;
                if  (IS_ERR (pipe_mnt) )
                        unregister_filesystem (&pipe_fs_type) ;
                else
                        err = 0;
        }
        return err;
}

static void __exit exit_pipe_fs (void)
{
        unregister_filesystem (&pipe_fs_type) ;
        mntput (pipe_mnt) ;
    }

module_init (init_pipe_fs)
module_exit (exit_pipe_fs)
```

pipefs 文件系统是作为一个模块来安装的，其中 module_init () 是模块的初始化函数，module_exit () 是模块的卸载函数，其更详细的解释将在第十章给出。

从 DECLARE_FSTYPE () 宏定义可以看出，pipefs 文件系统的 FS_NOMOUNT 和 FS_SINGLE 标志位为 1，这就意味着该文件系统不能从用户空间进行安装，并且在整个系统范围内只有一个超级块。FS_SINGLE 标志也意味着在通过 register_filesystem()成功地注册了该文件系统后，应该通过 kern_mount()来安装。

register_filesystem()函数把 pipe_fs_type 链接到 file_systems 链表，因此，你可以通过读/proc/filesystems 找到“pipefs”入口点，在那里，“nodev”标志表示没有设置 FS_REQUIRES_DEV 标志，即该文件系统没有对应的物理设备。

kern_mount()类似于 do_mount()，用来安装 pipefs 文件系统。当安装出现错误时，则调用 unregister_filesystem () 把 pipe_fs_type 从 file_systems 链表中拆除。

现在，pipefs 文件系统已被注册，并成为内核中的一个模块，从此我们就可以使用它了。pipefs 文件系统的入口点就是 pipe()系统调用，其内核实现函数为 sys_pipe()，而真正的工作是调用 do_pipe()函数来完成的，其代码在/fs/pipe.c 中，我们同时给出了对代码的注释。

```
int do_pipe (int *fd)
 {
        struct qstr this;
        char name[32];
        struct dentry *dentry;
        struct inode * inode;
            struct file *f1, *f2;   /*进程对每个已打开文件的操作是通过 file 结构进行的。
一个管道实际上就是一个存在于内存的文件，对这个文件的操作要通过两个已打开的文件进行，f1、f2 分别代表该管道的两端。*/
```

```
        int error;
         int i,j;

        error = -ENFILE;
       f1 = get_empty_filp () ;          /*管道两端各分配一个 file 结构*/

        if  (!f1)
                goto no_files;

        f2 = get_empty_filp () ;
        if  (!f2)
                goto close_f1;

                inode = get_pipe_inode () ;  /*每个文件都有一个 inode 结构。由于管道文件在
管道创建之前并不存在，因此，在创建管道时临时创建一个 inode 结构。*/

        if  (!inode)
                goto close_f12;

        error = get_unused_fd () ;       ／* 分配打开文件号*/
      if  (error < 0)
                goto close_f12_inode;
        i = error;

        error = get_unused_fd () ;
        if  (error < 0)
                goto close_f12_inode_i;
        j = error;

        error = -ENOMEM;
        sprintf (name, "[%lu]", inode->i_ino) ;
        this.name = name;
        this.len = strlen (name) ;
        this.hash = inode->i_ino; /* will go */
        dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &this);  /* File结构中有个指针 f_dentry
指向所打开文件的目录项 dentry 结构，而 dentry 中有个指针指向相应的 inode 结构。所以，调用 d_alloc()
分配一个目录项是为了把 file 结构与 inode 结构联系起来。*

        if  (!dentry)
                goto close_f12_inode_i_j;
       dentry->d_op = &pipefs_dentry_operations;

        d_add (dentry, inode) ;  ／*使已分配的 inode 结构与已分配的目录项结构挂勾*／

       f1->f_vfsmnt = f2->f_vfsmnt = mntget (mntget (pipe_mnt) ) ;   /* pipe_mnt 就是在
init_pipe_fs()中所获得的指向 vfsmount 结构的指针，因为这个结构多了两个使用者，因此调用两次
mntget()使其引用计数加 2 *／

       f1->f_dentry = f2->f_dentry = dget (dentry) ; ／*让两个已打开文件中的 f_dentry 指
针都指向这个目录项，并使目录项的引用计数加 1*／

                /* read file */
```

```
            f1->f_pos = f2->f_pos = 0;
            f1->f_flags = O_RDONLY;
            f1->f_op = &read_pipe_fops;
            f1->f_mode = 1;
            f1->f_version = 0;

            /* write file */
            f2->f_flags = O_WRONLY;
            f2->f_op = &write_pipe_fops;
            f2->f_mode = 2;
            f2->f_version = 0;

                fd_install（i, f1）;                  /*将已打开文件结构与分配得的打开文件
号相关联（打开文件号只在一个进程的范围内有效）。*/

            fd_install（j, f2）;
            fd[0] = i;    ／*使得 fd[0]为管道读出端的打开文件号*/
            fd[1] = j;    ／*使得 fd[1]为管道写出端的打开文件号*/

           return 0;
       ／*以下为释放各种资源*/
     close_f12_inode_i_j:
            put_unused_fd（j）;
     close_f12_inode_i:
            put_unused_fd（i）;
     close_f12_inode:
            free_page（（unsigned long） PIPE_BASE（*inode））;
            kfree（inode->i_pipe）;
            inode->i_pipe = NULL;
            iput（inode）;
     close_f12:
            put_filp（f2）;
     close_f1:
            put_filp（f1）;
    no_files:
           return error;
     }
```

下面对管道的单向性再做进一步的说明。从代码看出，把 f1 一端设置成“只读（O_RDONLY）”，另一端则设置成“只写（O_WRONLY）”。同时，两端的文件操作也分别设置成 read_pipe_fops 和 write_pipe_fops，其定义于 pipe.c 中：

```
        struct file_operations read_pipe_fops = {
            llseek:         pipe_lseek,
            read:           pipe_read,
            write:          bad_pipe_w,
            poll:           pipe_poll,
            ioctl:          pipe_ioctl,
            open:           pipe_read_open,
           release:       pipe_read_release,
     };

    struct file_operations write_pipe_fops = {
```

```
        llseek:         pipe_lseek,
        read:           bad_pipe_r,
        write:          pipe_write,
        poll:           pipe_poll,
        ioctl:          pipe_ioctl,
        open:           pipe_write_open,
        release:        pipe_write_release,
};
```

在 read_pipe_fops（）中的写操作函数为 bad_pipe_w（），而在 write_pipe_fops（）中的读操作函数为 bad_pipe_r（），这两个函数分别返回一个出错代码。尽管代表着管道两端的两个已打开文件一个只能读，一个只能写。但是，另一方面，这两个逻辑上已打开的文件指向同一个 inode，即用作管道的缓冲区，显然，这个缓冲区既支持读也支持写。这进一步说明了 file 、inode 及 dentry 之间的不同和联系。

8.6.2 磁盘文件系统 BFS

BFS 是 Berkeley fast File System 的简写，即柏克莱快速文件系统，是一种简单的基于磁盘的文件系统。BFS 将磁盘的分区分割为许多的柱面群，每一个柱面群依磁盘的大小，包含了 1～32 个相邻的柱面。BFS 模块的相关代码在 fs/bfs/inode.c 中：

```
static DECLARE_FSTYPE_DEV（bfs_fs_type, "bfs", bfs_read_super）;

static int __init init_bfs_fs（void）
{
        return register_filesystem（&bfs_fs_type）;
}

static void __exit exit_bfs_fs（void）
{
        unregister_filesystem（&bfs_fs_type）;
}

module_init（init_bfs_fs）
module_exit（exit_bfs_fs）
```

宏 DECLARE_FSTYPE_DEV()把 bfs_fs_type 文件类型的标志置为 FS_REQUIRES_DEV，表示 BFS 需要一个实际的块设备来进行安装。

模块的初始化函数调用 register_filesystem（）向 VFS 注册该文件系统，调用 unregister_filesystem（）注销该文件系统。

一旦文件系统被注册，我们就可以安装它，在安装一个文件系统时就会调用 fs_type->read_super()方法来读其超级块，具体到 BFS 文件系统则是调用 fs/bfs/inode.c 中的 bfs_read_super()函数，该函数的原型为：

```
    static struct super_block * bfs_read_super（struct super_block * s,
        void * data, int silent）
```

其中参数 s 是指向 super_block 的数据结构，在调用这个函数之前，该结构已被进行了一定的初始化，例如其 s_dev 域已经有了具体设备的设备号。但是，结构中的大部分内容还没有设置。而这个函数要做的工作就是从磁盘上读入该文件系统的超级块，并根据其内容设

置这个 super_block 数据结构。另一个指针 data 的使用，因文件系统而异，对于 BFS 文件系统来说并没有使用这个参数。而参数 silent，则表示在读超级块的过程中是否详细地报告出错信息。

现在，我们又回到在 VFS 级的调用函数 fs/super.c 中的 read_super()。在 read_super() 成功返回以后，VFS 就获得了对该文件系统模块的引用。

接下来，我们来考察以下在对该文件系统进行 I/O 操作时都发生些什么事情。我们已经考察过，iget(sb, ino)根据索引节点号从磁盘读入相应索引节点并在内存建立起相应的 inode 结构，在建立 inode 结构的过程中，还要做其他的事情，比如读 inode->i_op 和 inode->i_fop；打开一个文件就意味着把 inode->i_fop 拷贝到 file->f_op。

8.7　文件系统的系统调用

有关文件系统的系统调用有十几个，下面选其中几个简单地加以介绍。

8.7.1　open 系统调用

进程要访问一个文件，必须首先获得一个文件描述符，这是通过 open 系统调用来完成的。文件描述符是有限的资源，所以在不用时应该及时释放。

该系统调用是用来获得欲访问文件的文件描述符，如果文件并不存在，则还可以用它来创建一个新文件。其函数为 sys_open()，在 fs/open.c 中定义，函数如下：

```
asmlinkage long sys_open (const char * filename, int flags, int mode)
{
        char * tmp;
        int fd, error;

#if BITS_PER_LONG != 32
        flags |= O_LARGEFILE;
#endif
        tmp = getname (filename) ;
        fd = PTR_ERR (tmp) ;
        if  (!IS_ERR (tmp) )  {
                fd = get_unused_fd () ;
                if  (fd >= 0)  {
                        struct file *f = filp_open (tmp, flags, mode) ;
                        error = PTR_ERR (f) ;
                        if  (IS_ERR (f) )
                                goto out_error;
                        fd_install (fd, f) ;
                }
out:
                putname (tmp) ;
        }
        return fd;
```

```
out_error:
        put_unused_fd (fd) ;
        fd = error;
         goto out;
}
```

1．入口参数

（1）filename：　　　欲打开文件的路径。

（2）flags：　　　　规定如何打开该文件，它必须取下列 3 个值之一。

O_RDONLY　　　以只读方式打开文件

O_WRONLY　　　以只写方式打开文件

O_RDWR　　　　以读和写的方式打开文件

此外，还可以用或运算对下列标志值任意组合。

O_CREAT　　　　打开文件，如果文件不存在则建立文件

O_EXCL　　　　　如果已经置 O_CREAT 且文件存在，则强制 open()失败

O_TRUNC　　　　将文件的长度截为 0

O_APPEND　　　强制 write()从文件尾开始

对于终端文件，这 4 个标志是没有任何意义的，另提供了两个新的标志。

O_NOCTTY　　　停止这个终端作为控制终端

O_NONBLOCK　　使 open()、read()、write()不被阻塞。

这些标志的符号名称在/include/asmi386/fcntl.h 中定义。

（3）mode：这个参数实际上是可选的，如果用 open()创建一个新文件，则要用到该参数，它用来规定对该文件的所有者、文件的用户组和系统中其他用户的访问权限位。它用或运算对下列符号常量建立所需的组合。

S_IRUSR　　　　文件所有者的读权限位

S_IWUSR　　　　文件所有者的写权限位

S_IXUSR　　　　文件所有者的执行权限位

S_IRGRP　　　　文件用户组的读权限位

S_IWGRP　　　　文件用户组的写权限位

S_IXGRP　　　　文件用户组的执行权限位

S_IROTH　　　　文件其他用户的读权限位

S_IWOTH　　　　文件其他用户的写权限位

S_IXOTH　　　　文件其他用户的执行权限位

这些标志的符号名称在/include/linux/stat.h 中定义。

2．出口参数

返回一个文件描述符。

3．执行过程

sys_open()主要是调用 filp_open ()，这个函数也在 fs/open.c 中，这已在前面做过介绍。

从当前进程的 files_struct 结构的 fd 数组中找到第 1 个未使用项，使其指向 file 结构，将该项的下标作为文件描述符返回，结束。

在以上过程中，如果出错，则将分配的文件描述符、file 结构收回，inode 也被释放，函数返回一个负数以示出错，其中 PTR_ERR（）和 IS_ERR（）是出错处理函数，下一章将给予介绍。

8.7.2　read 系统调用

如果通过 open 调用获得一个文件描述符，而且是用 O_RDONLY 或 O_RDWR 标志打开的，就可以用 read 系统调用从该文件中读取字节。其内核函数在 fs/read_write.c 中定义：

```
asmlinkage ssize_t sys_read (unsigned int fd, char * buf, size_t count)
{
        ssize_t ret;
        struct file * file;

        ret = -EBADF;
        file = fget (fd) ;
        if  (file)  {
               if  (file->f_mode & FMODE_READ)  {
                      ret      =      locks_verify_area      (      FLOCK_VERIFY_READ,
file->f_dentry->d_inode,
                                          file, file->f_pos, count) ;
                      if  (!ret)  {
                             ssize_t (*read)  (struct file *, char *, size_t, loff_t *) ;
                             ret = -EINVAL;
                             if  (file->f_op &&  (read = file->f_op->read)  != NULL)
                                    ret = read (file, buf, count, &file->f_pos) ;
                      }
               }
               if  (ret > 0)
                      dnotify_parent (file->f_dentry, DN_ACCESS) ;
               fput (file) ;
        }
        return ret;
}
```

1．入口参数

（1）fd：要读的文件的文件描述符。

（2）buf：指向用户内存区中用来存储将读取字节的区域的指针。

（3）count：欲读的字节数。

2．出口参数

返回一个整数。在出错时返回-1；否则返回所读的字节数，通常这个数就是 count 值，但如果请求的字节数超过剩余的字节数，则返回实际读的字节数，例如文件的当前位置在文件尾，则返回值为 0。

3．执行过程

（1）函数 fget()根据打开文件号 fd 找到该文件已打开文件的 file 结构。

（2）取得了目标文件的 file 结构指针，并确认文件是以只读方式打开后，还要检查文件从当前位置 f_pos 开始的 count 个字节是否对读操作加上了“强制锁”，这是通过调用 locks_verify_area（）函数完成的，其代码在 fs.h 中。

（3）通过了对强制锁的检查后，就是读操作本身了。可想而知，不同的文件系统有不同的读操作，具体的文件系统通过 file_operations 结构提供用于读操作的函数指针。就 Ext2 文件系统来说，它有两个这样的结构，一个是 Ext2_file_operations，另一个是 Ext2_dir_operations，视操作的目标为文件或目录而选择其一，在打开文件时，操作结构就安装在其 file 结构中。对于常规文件，这个函数指针指向 generic_file_read（），其代码在 mm/filemap.c 中。

（4）如果读操作的返回值大于 0，说明出错，则调用 dnotify_parent（）报告错误，并释放文件描述符、file 结构、inode 结构。

8.7.3 fcntl 系统调用

这个系统调用功能比较多，可以执行多种操作，其内核函数在 fs/fcntl.c 中定义。

1．入口参数

（1）fd：欲访问文件的文件描述符。

（2）cmd：要执行的操作的命令，这个参数定义了 10 个标志，下面介绍其中的 5 个，F_DUPFD、F_GETFD、F_SETFD、F_GETFL 和 F_SETFL

（3）arg：可选，主要根据 cmd 来决定是否需要。

2．出口参数：根据第二个参数（cmd）的不同，这个返回值也不一样

3．函数功能

如果第二个参数（cmd）取值是 F_DUPFD，则进行复制文件描述符的操作。它需要用到第三个参数 arg，这时 arg 是一个文件描述符，fcntl(fd，F_DUPFD，arg)在 files_struct 结构中从指定的 arg 开始搜索空闲的文件描述符，找到第一个后，将 fd 的内容复制进来，然后将新找到的文件描述符返回。

第二个参数（cmd）取值是 F_GETFD，则返回 files_struct 结构中 close_on_exec 的值。

无需第三个参数。

第二个参数（cmd）取值是 F_SETFD，则需要第三个参数，若 arg 最低位为 1，则对 close_on_exec 置位，否则清除 close_on_exec。

第二个参数（cmd）取值是 F_GETFL，则用来读取 open 系统调用第二个参数设置的标志，即文件的打开方式（O_RDONLY，O_WRONLY，O_APPEND 等），它不需要第三个参数。实际上上这时函数返回的是 file 结构中的 flags 域。

第二个参数（cmd）取值是 F_SETFL，则用来对 open 系统调用第二个参数设置的标志进行改变，但是它只能对 O_APPEND 和 O_NONBLOCK 标志进行改变，这时需要第三个参数 arg，用来确定如何改变。函数返回 0 表示操作成功，否则返回-1，并置一个错

8.8　Linux 2.4 文件系统的移植问题

Linux 内核源代码 2.2.x 版本及 2.4.x 版本之间有一定的变化，我们把 2.2.x 版本叫旧版，把 2.4.x 叫新版。下面给出文件系统的变化。

1．模块处理方式发生了变化

模块的初始化方式有所变化，在旧版本中的初始化方式如下（我们把某个文件系统叫做 myfs）：

```
static struct file_system_type myfs_fs_type =
      { "myfs", FS_REQUIRES_DEV, myfs_read_super, NULL };

__initfunc（int init_myfs_fs（void））{
   return register_filesystem（&myfs_fs_type）;
}

  #ifdef MODULE
  EXPORT_NO_SYMBOLS;

   int init_module（void） {
       return init_myfs_fs（）;
  }

  void cleanup_module（void） {
       unregister_filesystem（&myfs_fs_type）;
  }
  #endif
```

另外，MOD_INC_USE_COUNT 宏在文件系统的 read_super()中被调用，而 MOD_DEC_USE_COUNT 宏在 put_super()中被调用。新版中的初始化方法如下：

```
static DECLARE_FSTYPE_DEV（myfs_fs_type, "myfs", myfs_read_super）;

static int __init init_myfs_fs（void） {
   return register_filesystem（&myfs_fs_type）;
}
```

```
static void __exit exit_myfs_fs (void)  {
   unregister_filesystem (&myfs_fs_type) ;
}

EXPORT_NO_SYMBOLS;
module_init (init_myfs_fs) ;
module_exit (exit_myfs_fs) ;
```

MOD_XXX_USE_COUNT 现在由 VFS 文件系统在注册时进行处理。从代码可以看出，新版本的处理方式更具可读性，而本质上并没有多大变化。

2．大文件的支持（Large File Support，LFS）

VFS 现在支持 64 位文件（仅适用于 x86 和 Sparc 平台）：

- 使用 64 位类型 loff_t
- 但内核还不支持 64 位的 getrlimit()和 setrlimit()系统调用；
- glibc 库支持 getrlimit64() 和 setrlimit64 ()。

3．新的错误处理方式

旧版中错误处理如下：

```
if  (!dir || !dir->i_nlink)  {
   *err = -EPERM;
   return NULL;
   }
```

在新版中，对错误的处理调用了 ERR_PTR()，PTR_ERR() 和 IS_ERR()函数：

```
if  (!dir || !dir->i_nlink)
    return ERR_PTR (-EPERM) ;
```

这几个错误处理函数定义于 include/linux/fs.h 中：

```
static inline void *ERR_PTR (long error)
{
        return  (void *)  error;
}

static inline long PTR_ERR (const void *ptr)
{
        return  (long)  ptr;
}

static inline long IS_ERR (const void *ptr)
{
        return  (unsigned long) ptr >  (unsigned long) -1000L;
}
```

4．inode 结构

在旧版中，file_operations 在 inode_operations 结构中定义，而在新版中已移到 inode 结构中，即：

```
struct file_operations *i_fop;
```

其引用形式为：inode->i_fop。

另外，inode 中 count 类型的定义也有所改变：

旧版：int i_count;

新版：atomic_t i_count;

这种类型的定义是与体系结构独立的，因此，对这些变量的访问就是原子操作，例如对变量 v 的读和设置函数为：

```
atomic_read (v) ;
atomic_set (v, value) ;
```

这种形式也应用于 file 结构 和 dentry 结构。

5．dentry 高速缓存

在旧版中，删除一个 dentry 的函数形式为：

```
void (*d_delete) (struct dentry *) ;
```

在新版中，删除一个 dentry 的函数形式为：

```
int (*d_delete) (struct dentry *) ;
```

返回一个整数表示删除的成功与否。

在旧版中，分配一个根目录项的函数形式为：

```
struct dentry *d_alloc_root (struct inode *, struct dentry *) ;
```

在新版中，分配一个根目录项的函数形式为：

```
struct dentry *d_alloc_root (struct inode *) ;
```

6．VFS 操作

在旧版中，filldir 帮助函数的形式为：

```
typedef int (*filldir_t) (void *, const char *, int, off_t, ino_t) ;
```

在新版中，filldir 帮助函数的形式为：

```
typedef int (*filldir_t) (void *, const char *, int, off_t, ino_t, unsigned) ;
```

新增加的参数表示文件类型，其定义于 fs.h 中：

```
/*
 * File types
 */
#define DT_UNKNOWN     0
#define DT_FIFO        1
#define DT_CHR         2
#define DT_DIR         4
#define DT_BLK         6
#define DT_REG         8
#define DT_LNK         10
#define DT_SOCK        12
#define DT_WHT         14
```

7．各种操作结构的指定方式发生变化

所有操作结构的指定方式发生了变化，旧版中的形式为：

```
struct file_operations myfs_file_operations = {
   myfs_file_lseek,
   generic_file_read,
```

```
        generic_file_write,
        NULL,
        NULL,
        myfs_ioctl,
        NULL,
    };
```

新版中的形式为：

```
    struct file_operations myfs_file_operations = {
        llseek: myfs_file_lseek,
        read: generic_file_read,
        write: generic_file_write,
        ioctl: myfs_ioctl,
    };
```

这种形式使用了 GNU C 语言的扩展形式，符合 ISO C99 标准，C99 标准指定的初始化者的形式为：

```
    struct foo {
        int foo;
        long bar;
    };

    struct foo x = { .bar = 3, .foo = 4 };
```

8．VFS 的 file_operations

在 fsync（）函数中增加了一个新的参数；如果这个参数被设置，则不干预对时间标记的刷新：

旧版：int (*fsync) (struct file *, struct dentry *);

新版：int (*fsync) (struct file *, struct dentry *, int);

另外，原来 file_operations 中的两个操作：

```
int  (*check_media_change)   (kdev_t dev)；
         int  (*revalidate)   (kdev_t dev)；
```

被移到一个新的结构 block_device_operations：

```
    struct block_device_operations {
        int  (*open)   (struct inode *, struct file *)；
        int  (*release)   (struct inode *, struct file *)；
        int  (*ioctl)  (struct inode *, struct file *, unsigned, unsigned long)；
        int  (*check_media_change)  (kdev_t)；
        int  (*revalidate)   (kdev_t)；
    };
```

这个新的结构成为 inode 结构中一个新的域：

新版：struct block_device *i_bdev;

于是在 block_device 中有一个指向这个操作结构的指针：

```
    struct block_device {
        struct list_head bd_hash;
        atomic_t bd_count;
        dev_t bd_dev;
        atomic_t bd_openers;
```

```
        const struct
        block_device_operations *bd_op;
        struct semaphore bd_sem;
    };
```

另外，file_operations（）中还增加了以下两个函数。在不用保持大内核锁的情况下，所有文件系统都可以调用这两个函数。这两个函数还实现了 readv() 和 writev()系统调用。

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *) ;
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *) ;
```

9．VFS 的 inode_operations

file_operations 指针从 inode_operations 结构移到 inode 结构。

inode_operations 结构中，follow_link（）函数的形式也发生了变化：

旧版中：struct dentry * (*follow_link) (struct dentry *, struct dentry *, unsigned int);

新版中：int (*follow_link) (struct dentry *, struct nameidata *);

从调用形式可以看出，第一个参数仍然相同，新版参数的 nameidata 结构中包含了旧版中的最后两个参数：

```
    struct nameidata {
      struct dentry *dentry;
      struct vfsmount *mnt;
      struct qstr last;
      unsigned int flags;
      int last_type;
    };
```

inode_operations 结构中还增加了以下两个函数：

```
int (*setattr) (struct dentry *, struct iattr *) ;
int (*getattr) (struct dentry *, struct iattr *) ;
```

这两个函数（实际上仅仅是 setattr()）取代了旧版中 superblock 操作中的 notify_change()函数。另外，下列 5 个函数已经全部被取消：

```
int (*readpage) (struct file *, struct page *) ;
int (*writepage) (struct file *, struct page *) ;
int (*updatepage) (struct file *, struct page *, unsigned long, unsigned int, int) ;
int (*bmap) (struct inode *,int) ;
int (*smap) (struct inode *,int) ;
```

10．VFS 的 super_operations

在 super_operations 结构中，write_inode（）函数的形式有所变化：

旧版中：void (*write_inode) (struct inode *);

新版中：void (*write_inode) (struct inode *, int);

新增加的参数是一个布尔标志，用来决定是否把 inode 同步地写到磁盘。

相反，statfs() 函数中少一个参数，因为 statfs 结构的大小是没有必要的。

旧版中：int (*statfs) (struct super_block *, struct statfs *, int);

新版中：int (*statfs) (struct super_block *, struct statfs *);

最后，notify_change（）函数被 getattr()和 setattr()函数取代。

第九章　Ext2 文件系统

Ext2（第二扩充文件系统）是一种功能强大、易扩充、性能上进行了全面的优化的文件系统，也是当前 Linux 文件系统实际上的标准。

Linux 的第一个文件系统是 Minix，它原本是为 Minix 这个操作系统所使用的，Linus Torvalds 写 Linux 之前，学习的就是 Tanenbaum 所写的《Operating Systems Design And Implementation》，该书以 Tanenbaum 自己写的 Minix 系统为例，所以 Linus 就将 Minix 文件系统改写后用于 Linux，但这个文件系统有几个主要的缺限：

- 磁盘分区大小必须小于 64MB；
- 必须使用 14 个字符定长的文件名；
- 难于扩展。

在 VFS 被加入内核后，1992 年 4 月，第一个专门为 Linux 所写的文件系统 Ext（扩充文件系统）被加入了 0.96c 这个版本。Ext 使 Minix 的缺陷得以改进，一是它最大可支持 2GB 的磁盘分区，二是其文件名最长可达 255 个字符。但它仍有自己的缺陷：它使用链表管理未分配的数据块和节点，这样当文件系统投入使用后，链表变得杂乱无序，文件系统中会产生很多碎片。

1993 年，Remy Card 对 Ext 做出了改进，写成了 Ext2。Ext2 有如下几方面的特点。

- 它的节点中使用了 15 个数据块指针，这样它最大可支持 4TB 的磁盘分区。
- 它使用变长的目录项，这样既可以不浪费磁盘空间，又能支持最长 255 个字符的文件名。
- 使用位图来管理数据块和节点的使用情况，解决了 Ext 出现的问题。
- 最重要的一点是，它在磁盘上的布局做了改进，即使用了块组的概念，从而使数据的读和写更快、更有效，也便系统变得更安全可靠。
- 易于扩展。

9.1　基本概念

在上一章中，我们把 Ext2、Minix、Ext 等实际可使用的文件系统称为具体文件系统。具体文件系统管理的是一个逻辑空间，这个逻辑空间就像一个大的数组，数组的每个元素就是文件系统操作的基本单位——逻辑块，逻辑块是从 0 开始编号的，而且，逻辑块是连续的。与逻辑块相对的是物理块，物理块是数据在磁盘上的存取单位，也就是每进行一次 I/O 操作，最小传输的数据大小。我们知道数据是存储在磁盘的扇区中的，那么扇区是不是物理块呢？或者物理块是多大呢？这涉及到文件系统效率的问题。

如果物理块定的比较大，比如一个柱面大小，这时，即使是 1 个字节的文件都要占用整个一个柱面，假设 Linux 环境下文件的平均大小为 1KB，那么分配 32KB 的柱面将浪费 97%的磁盘空间，也就是说，大的存取单位将带来严重的磁盘空间浪费。另一方面，如果物理块过小，则意味着对一个文件的操作将进行更多次的寻道延迟和旋转延迟，因而读取由小的物理块组成的文件将非常缓慢！可见，时间效率和空间效率在本质上是相互冲突的。

因此，最优的方法是计算出 Linux 环境下文件的平均大小，然后将物理块大小定为最接近扇区的整数倍大小。在 Ext2 中，物理块的大小是可变化的，这取决于你在创建文件系统时的选择，之所以不限制大小，也正体现了 Ext2 的灵活性和可扩充性，一是因为要适应近年来文件的平均长度缓慢增长的趋势，二是为了适应不同的需要。比如，如果一个文件系统主要用于 BBS 服务，考虑到 BBS 上的文章通常很短小，所以，物理块选得小一点是恰当的。通常，Ext2 的物理块占一个或几个连续的扇区，显然，物理块的数目是由磁盘容量等硬件因素决定的。逻辑块与物理块的关系类似于虚拟内存中的页与物理内存中的页面的关系。

具体文件系统所操作的基本单位是逻辑块，只在需要进行 I/O 操作时才进行逻辑块到物理块的映射，这显然避免了大量的 I/O 操作，因而文件系统能够变得高效。逻辑块作为一个抽象的概念，它必然要映射到具体的物理块上去，因此，逻辑块的大小必须是物理块大小的整数倍，一般说来，两者是一样大的。

通常，一个文件占用的多个物理块在磁盘上是不连续存储的，因为如果连续存储，则经过频繁的删除、建立、移动文件等操作，最后磁盘上将形成大量的空洞，很快磁盘上将无空间可供使用。因此，必须提供一种方法将一个文件占用的多个逻辑块映射到对应的非连续存储的物理块上去，Ext2 等类文件系统是用索引节点解决这个问题的，具体实现方法后面再予以介绍。

为了更好地说明逻辑块和物理块的关系，我们来看一个例子。

假设用户要对一个已有文件进行写操作，用户进程必须先打开这个文件，file 结构记录了该文件的当前位置。然后用户把一个指向用户内存区的指针和请求写的字节数传送给系统，请求写操作，这时系统要进行两次映射。

（1）一组字节到逻辑块的映射。

这个映射过程就是找到起始字节到结束字节所占用的所有逻辑块号。这是因为在逻辑空间，文件传输的基本单位是逻辑块而不是字节。

（2）逻辑块到物理块的映射。

这个过程必须要用到索引节点结构，该结构中有一个物理块指针数组，以逻辑块号为索引，通过这些指针找到磁盘上的物理块，具体实现将在介绍 Ext2 索引节点时再进行介绍。

图 9.1 是由一组请求的字节到物理块的映射过程示意图。

有了逻辑块和物理块的概念，我们也就知道通常所说的数据块是指逻辑块，以下没有特别说明，块或数据块指的是逻辑块。

在 Ext2 中，还有一个重要的概念：片（fragment），它的作用是什么？

每个文件必然占用整数个逻辑块，除非每个文件大小都恰好是逻辑块的整数倍，否则最后一个逻辑块必然有空间未被使用，实际上，每个文件的最后一个逻辑块平均要浪费一半的空间，显然最终浪费的还是物理块。在一个有很多文件的系统中，这种浪费是很大的。Ext2 使用片来解决这个问题。

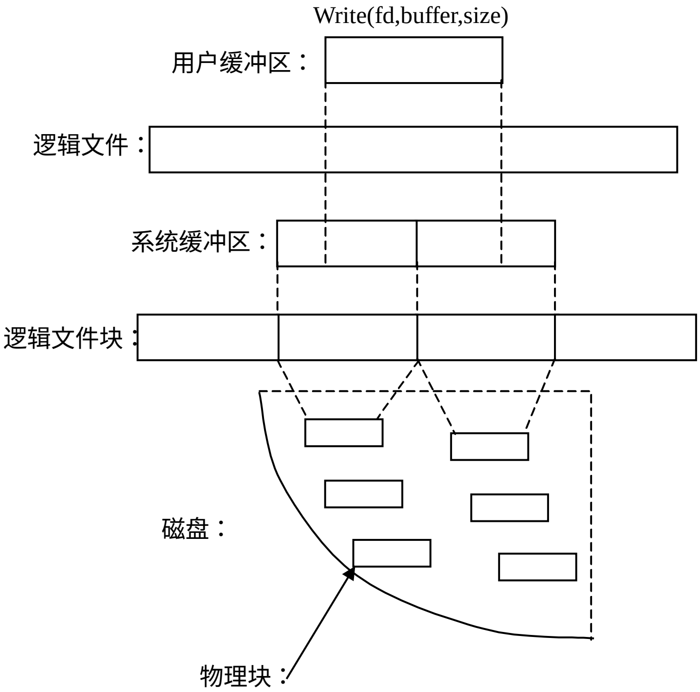


图 9.1 一组字节映射到物理块的示意图

片也是一个逻辑空间中的概念，其大小在 1KB 至 4KB 之间，但片的大小总是不大于逻辑块。假设逻辑块大小为 4KB，片大小为 1KB，物理块大小也是 1KB，当你要创建一个 3KB 大小的文件时，实际上分配给你了 3 个片，而不会给你一个逻辑块，当文件大小增加到 4KB 时，文件系统则分配一个逻辑块给你，而原来的四个片被清空。如果文件又增加到 5KB 时，则占用 1 个逻辑块和 1 个片。上述 3 种情况下，所占用的物理块分别是 3 个、4 个、5 个，如果不采用片，则要用到 4 个、4 个、8 个物理块，可见，使用片，减少了磁盘空间的浪费。当然，在物理块和逻辑块大小一样时，片就没有意义了。

由上面分析也可看出：

物理块大小<=片大小<=逻辑块大小

9.2 Ext2 的磁盘布局和数据结构

9.2.1 Ext2 的磁盘布局

文件系统的逻辑空间最终要通过逻辑块到物理块的映射转化为磁盘等介质上的物理空间，因此，对逻辑空间的组织和管理的好坏必然影响到物理空间的使用情况。一个文件系统，在磁盘上如何布局，要综合考虑以下几个方面的因素。

- 首先也是最重要的是要保证数据的安全性，也就是说当在向磁盘写数据时发生错误，要能保证文件系统不遭到破坏。
- 其次，数据结构要能高效地支持所有的操作。Ext2 中，最复杂的操作是硬链接操作。硬链接允许一个文件有多个名称，通过任何一个名称都将访问相同的数据。另一个比较复杂的操作是删除一个已打开的文件。
- 第三，磁盘布局应使数据查找的时间尽量短，以提高效率。驱动器查找分散的数据要比查找相邻的数据花多得多的时间。一个好的磁盘布局应该让相关的数据尽量连续分布。例如，同一个文件的数据应连续分布，并和包含该文件的目录文件相邻。
- 最后，磁盘布局应该考虑节省空间。虽然现在节省磁盘空间已不太重要，但也不应该无谓地浪费磁盘空间。

Ext2 的磁盘布局可以说综合考虑了以上几方面的因素，因此，它是一种高效、安全的文件系统。图 9.2 是 Ext2 的磁盘布局在逻辑空间中的映像。可以看出，它由一个引导块和重复的块组构成的，每个块组又由超级块、组描述符表、块位图、索引节点位图、索引节点表、数据区构成。引导块中含有可执行代码，启动计算机时，硬件从引导设备将引导块读入内存，然后执行它的代码。系统启动后，引导块不再使用。因此，引导块不属于文件系统管理。

以下对块组及块组中的数据结构的介绍可以充分领略 Ext2 磁盘布局的优越之处。

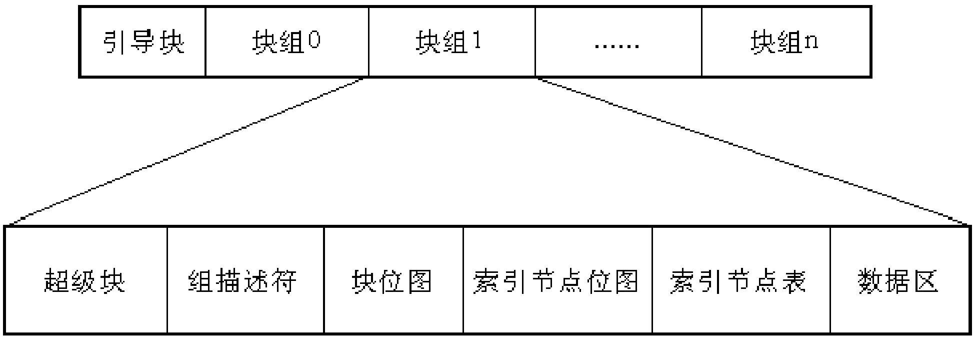


图 9.2 Ext2 磁盘布局在逻辑空间的映像

9.2.2 Ext2 的超级块

Ext2 超级块是用来描述 Ext2 文件系统整体信息的数据结构，是 Ext2 的核心所在。它是一个 ext2_super_block 数据结构（在 include/Linux/ext2_fs.h 中定义），其各个域及含义如下。

```
struct ext2_super_block
{     __u32     s_inodes_count;              /*文件系统中索引节点总数 */
      __u32     s_blocks_count;              /*文件系统中总块数 */
      __u32     s_r_blocks_count;        /*为超级用户保留的块数 */
      __u32     s_free_blocks_count;     /*文件系统中空闲块总数 */
      __u32     s_free_inodes_count;     /*文件系统中空闲索引节点总数*/
      __u32     s_first_data_block;          /* 文件系统中第一个数据块 */
      __u32     s_log_block_size;        /* 用于计算逻辑块大小 */
```

```
    __s32      s_log_frag_size;         /* 用于计算片大小 */
    __u32      s_blocks_per_group;          /* 每组中块数 */
    __u32      s_frags_per_group;       /* 每组中片数 */
    __u32      s_inodes_per_group;          /* 每组中索引节点数 */
    __u32      s_mtime;                 /*最后一次安装操作的时间 */
    __u32      s_wtime;                 /*最后一次对该超级块进行写操作的时间 */
    __u16      s_mnt_count;             /* 安装计数 */
    __s16      s_max_mnt_count;         /* 最大可安装计数 */
    __u16      s_magic;                 /* 用于确定文件系统版本的标志 */
    __u16      s_state;                 /* 文件系统的状态*/
    __u16      s_errors;                /* 当检测到有错误时如何处理 */
    __u16      s_minor_rev_level;           /* 次版本号 */
    __u32      s_lastcheck;             /* 最后一次检测文件系统状态的时间 */
    __u32      s_checkinterval;         /* 两次对文件系统状态进行检测的间隔时间 */
    __u32      s_rev_level;             /* 版本号 */
    __u16      s_def_resuid;            /* 保留块的默认用户标识号 */
    __u16      s_def_resgid;            /* 保留块的默认用户组标识号*/

  /*
   * These fields are for EXT2_DYNAMIC_REV superblocks only.
   *
   * Note: the difference between the compatible feature set and
   * the incompatible feature set is that if there is a bit set
   * in the incompatible feature set that the kernel doesn't
   * know about, it should refuse to mount the filesystem.
   *
   * e2fsck's requirements are more strict; if it doesn't know
   * about a feature in either the compatible or incompatible
   * feature set, it must abort and not try to meddle with
   * things it doesn't understand...
   */
 __u32    s_first_ino;                  /* 第一个非保留的索引节点 */
 __u16    s_inode_size;                 /* 索引节点的大小 */
   __u16    s_block_group_nr;           /* 该超级块的块组号 */
   __u32    s_feature_compat;           /* 兼容特点的位图*/
   __u32    s_feature_incompat;         /* 非兼容特点的位图 */
   __u32    s_feature_ro_compat;        /* 只读兼容特点的位图*/
   __u8     s_uuid[16];                 /* 128 位的文件系统标识号*/
   char     s_volume_name[16];          /* 卷名 */
   char     s_last_mounted[64];         /* 最后一个安装点的路径名 */
   __u32    s_algorithm_usage_bitmap;   /* 用于压缩*/
    /*
    * Performance hints.  Directory preallocation should only
    * happen if the EXT2_COMPAT_PREALLOC flag is on.
    */
   __u8     s_prealloc_blocks;          /* 预分配的块数*/
   __u8     s_prealloc_dir_blocks;      /* 给目录预分配的块数 */
   __u16    s_padding1;
   __u32    s_reserved[204];            /* 用 NULL 填充块的末尾 */
  };
```

从中我们可以看出，这个数据结构描述了整个文件系统的信息，下面对其中一些域作一些解释。

（1）文件系统中并非所有的块普通用户都可以使用，有一些块是保留给超级用户专用的，这些块的数目就是在 s_r_blocks_count 中定义的。一旦空闲块总数等于保留块数，普通用户无法再申请到块了。如果保留块也被使用，则系统就可能无法启动了。有了保留块，我们就可以确保一个最小的空间用于引导系统。

（2）逻辑块是从 0 开始编号的，对块大小为 1KB 的文件系统，s_first_data_block 为 1，对其他文件系统，则为 0。

（3）s_log_block_size 是一个整数，以 2 的幂次方表示块的大小，用 1024 字节作为单位。因此，0 表示 1024 字节的块，1 表示 2048 字节的块，如此等等。

同样,片的大小计算方法也是类似的,因为 Ext2 中还没有实现片,因此,s_log_frag_size 与 s_log_block_size 相等。

（4）Ext2 要定期检查自己的状态，它的状态取下面两个值之一。

```
#define EXT2_VALID_FS     0x0001
```

文件系统没有出错。

```
#define EXT2_ERROR_FS    0x0002
```

内核检测到错误。

s_lastcheck 就是用来记录最近一次检查状态的时间，而 s_checkinterval 则规定了两次检查状态的最大允许间隔时间。

（5）如果检测到文件系统有错误，则对 s_errors 赋一个错误值。一个好的系统应该能在错误发生时进行正确处理，有关 Ext2 如何处理错误将在后面介绍。

超级块被读入内存后，主要用于填写 VFS 的超级块，此外，它还要用来填写另外一个结构，这就是 ext2_super_info 结构，这一点我们可以从有关 Ext2 超级块的操作中看出，比如 ext2_read_super()。之所以要用到这个结构，是因为 VFS 的超级块必须兼容各种文件系统的不同的超级块结构,所以对某个文件系统超级块自己的特性必须用另一个结构保存于内存中，以加快对文件的操作，比如对 Ext2 来说，片就是它特有的，所以不能存储在 VFS 超级块中。Ext2 中的这个结构是 ext2_super_info，它其中的信息多是从磁盘上的索引节点计算得来的 。该结构定义于 include/Linux/ext2_fs_sb.h，下面是该结构及各个域含义：

```
struct ext2_sb_info
{
    unsigned long s_frag_size;              /* 片大小（以字节计） */
    unsigned long s_frags_per_block;        /* 每块中片数 */
    unsigned long s_inodes_per_block;       /* 每块中节点数*/
    unsigned long s_frags_per_group;        /* 每组中片数*/
    unsigned long s_blocks_per_group;       /* 每组中块数 */
    unsigned long s_inodes_per_group;       /*每组中节点数 */
    unsigned long s_itb_per_group;          /* 每组中索引节点表所占块数 */
    unsigned long s_db_per_group;           /* 每组中组描述符所在块数 */
    unsigned long s_desc_per_block;         /* 每块中组描述符数 */
    unsigned long s_groups_count;           /* 文件系统中块组数 */
    struct buffer_head * s_sbh;             /* 指向包含超级块的缓存 */
    struct buffer_head ** s_group_desc;   /*指向高速缓存中组描述符表块的
                                            指针数组的一个指针*/
```

```
    unsigned short s_loaded_inode_bitmaps;     /* 装入高速缓存中的节点位图块数*/
    unsigned short s_loaded_block_bitmaps;     /*装入高速缓存中的块位图块数*/
    unsigned long s_inode_bitmap_number[Ext2_MAX_GROUP_LOADED];
    struct buffer_head * s_inode_bitmap[Ext2_MAX_GROUP_LOADED];
    unsigned long s_block_bitmap_number[Ext2_MAX_GROUP_LOADED];
    struct buffer_head * s_block_bitmap[Ext2_MAX_GROUP_LOADED];
    int s_rename_lock;                         /*重命名时的锁信号量*/
    struct wait_queue * s_rename_wait;         /*指向重命名时的等待队列*/
    unsigned long  s_mount_opt;                /*安装选项*/
    unsigned short s_resuid;                   /*默认的用户标识号*/
    unsigned short s_resgid;                   /* 默认的用户组标识号*/
    unsigned short s_mount_state;              /*专用于管理员的安装选项*/
    unsigned short s_pad;                          /*填充*/
    int s_inode_size;                          /*节点的大小*/
    int s_first_ino;                               /*第一个节点号*/
};
```

s_block_bitmap_number[] 、 s_block_bitmap[] 、 s_inode_bitmap_number[] 、s_inode_bitmap[]是用来管理位图块高速缓存的，在介绍位图时再作说明。

另外，由于每个文件系统的组描述符表可能占多个块，这些块进入缓存后，用一个指针数组分别指向它们在缓存中的地址，而 s_group_desc 则是用来指向这个数组的，用相对于组描述符表首块的块数作索引，就可以找到指定的组描述符表块。

图 9.3 是 3 个与超级块相关的数据结构的关系示意图。

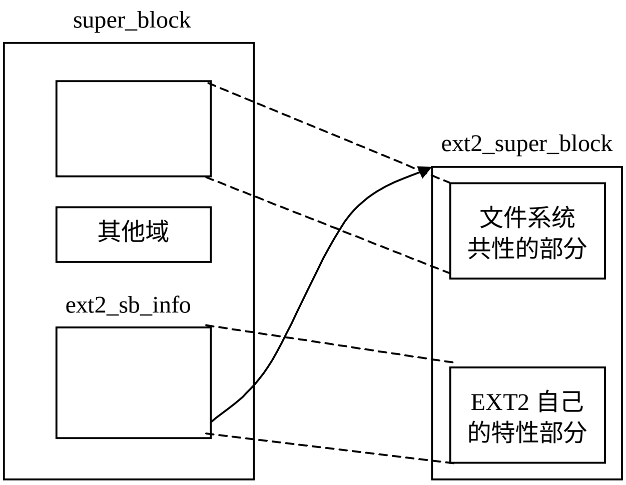


图 9.3 三种超级块结构的关系示意图

9.2.3 Ext2 的索引节点

Ext2 和 UNIX 类的文件系统一样，使用索引节点来记录文件信息。每一个普通文件和目录都有唯一的索引节点与之对应，索引节点中含有文件或目录的重要信息。当你要访问一个文件或目录时，通过文件或目录名首先找到与之对应的索引节点，然后通过索引节点得到文件或目录的信息及磁盘上的具体的存储位置。Ext2 的索引节点的数据结构叫 ext2_inode，在

include/Linux/ext2_fs.h 中定义，下面是其结构及各个域的含义（不同版本，该结构略有不同）。

```
struct ext2_inode {
    __u16 i_mode;                   /* 文件类型和访问权限 */
     __u16    i_uid;                    /* 文件拥有者标识号*/
     __u32    i_size;                   /* 以字节计的文件大小 */
     __u32    i_atime;              /* 文件的最后一次访问时间 */
     __u32    i_ctime;              /* 该节点最后被修改时间 */
     __u32    i_mtime;              /* 文件内容的最后修改时间 */
     __u32    i_dtime;              /* 文件删除时间 */
     __u16    i_gid;                    /* 文件的用户组标志符 */
     __u16    i_links_count;        /* 文件的硬链接计数 */
     __u32    i_blocks;             /* 文件所占块数（每块以 512 字节计）*/
     __u32    i_flags;              /* 打开文件的方式 */
   union                            /*特定操作系统的信息*/
  __u32  i_block[Ext2_N_BLOCKS];    /* 指向数据块的指针数组 */
     __u32    i_version;                /* 文件的版本号（用于 NFS） */
     __u32    i_file_acl;               /*文件访问控制表（已不再使用） */
     __u32    i_dir_acl;                /*目录访问控制表（已不再使用）*/
      __u8    l_i_frag;             /* 每块中的片数 */
     __u32   i_faddr;               /* 片的地址 */
      union                         /*特定操作系统信息*/
}
```

从中可以看出，索引节点是用来描述文件或目录信息的。

以下，对其中一些域作一定解释。

（1）前面说过，Ext2 通过索引节点中的数据块指针数组进行逻辑块到物理块的映射。在 Ext2 索引节点中，数据块指针数组共有 15 项，前 12 个为直接块指针，后 3 个分别为“一次间接块指针”、“二次间接块指针”、“三次间接块指针”，如图 9.4 所示。

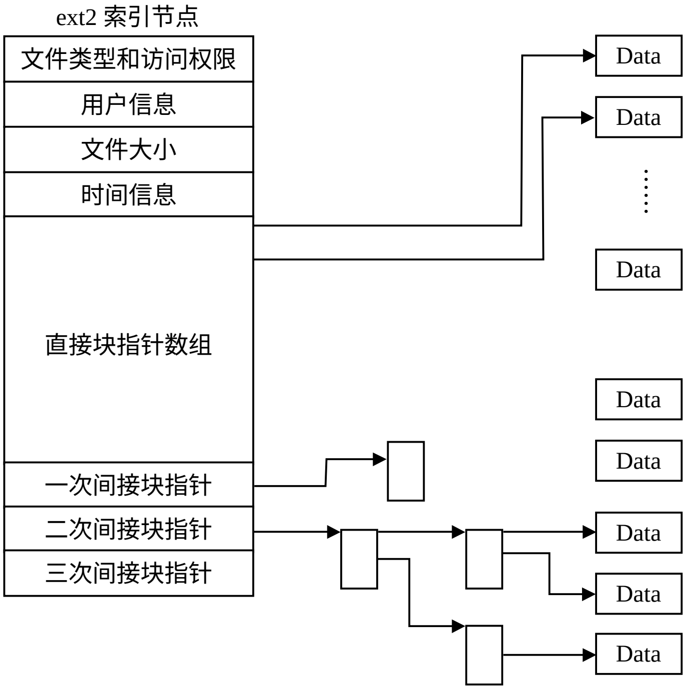


图 9.4 索引节点的数据块指针数组示意图

所谓“直接块”，是指该块直接用来存储文件的数据，而“一次间接块”是指该块不存储数据，而是存储直接块的地址，同样，“二次间接块”存储的是“一次间接块”的地址。这里所说的块，指的都是物理块。Ext2 默认的物理块大小为 1KB，块地址占 4 个字节（32 位），所以每个物理块可以存储 256 个地址。这样，文件大小最大可达 12KB+256KB+ 64MB+16GB。但实际上，Linux 是 32 位的操作系统，故文件大小最大只能为 4GB。

系统是以逻辑块号为索引查找物理块的。例如，要找到第 100 个逻辑块对应的物理块，因为 256+12>100+12，所以要用到一次间接块，在一次间接块中查找第 88 项，此项内容就是对应的物理块的地址。而如果要找第 1000 个逻辑块对应的物理块，由于 1000>256+12，所以要用到二次间接块了。

（2）索引节点的标志（flags）取下列几个值的可能组合。

EXT2_SECRM_FL　0x00000001

完全删除标志。设置这个标志后，删除文件时，随机数据会填充原来的数据块。

EXT2_UNRM_FL　0x00000002

可恢复标志。设置这个标志后，删除文件时，文件系统会保留足够信息，以确保文件仍能恢复（仅在一段时间内）。

EXT2_COMR_FL　0x00000004

压缩标志。设置这个标志后，表明该文件被压缩过。当访问该文件时，文件系统必须采用解压缩算法进行解压。

EXT2_SYNC_FL　0x00000008

同步更新标志。设置该标志后，则该文件必须和内存中的内容保持一致，对这种文件进行异步输入、输出操作是不允许的。这个标志仅用于节点本身和间接块。数据块总是异步写

入磁盘的。

除了这几个常用标志外，还有 12 个标志就不一一介绍了。

（3）索引节点在磁盘上是经过编号的。其中，有一些节点有特殊用途，用户不能使用。这些特殊节点也在 include/Linux/ext2_fs.h 中定义。

```
#define EXT2_BAD_INO              1
```

该节点所对应的文件中包含着该文件系统中坏块的链接表。

```
#define EXT2_ROOT_INO             2
```

该文件系统的根目录所对应的节点。

```
#define EXT2_IDX_INO          3
```

ACL（访问控制链表）节点。

```
#define EXT2_DATA_INO         4
```

ACL 节点。

```
#define EXT2_BOOT_LOADER_INO 5
```

用于引导系统的文件所对应的节点。

```
#define EXT2_UNDEL_DIR_INO      6
```

文件系统中可恢复的目录对应的节点。

没有特殊用途的第一个节点号为 11。

```
#define  EXT2_FIRST_INO         11
```

（4）文件的类型、访问权限、用户标识号、用户组标识号等将在后面介绍。

与 Ext2 超级块类似，当磁盘上的索引节点调入内存后，除了要填写 VFS 的索引节点外，系统还要根据它填写另一个数据结构，该结构叫 ext2_inode_info，其作用也是为了存储特定文件系统自己的特性，它在 include/Linux/ext2_fs_i.h 中定义如下：

```
struct ext2_inode_info
{
     __u32     i_data[15];                  /*数据块指针数组*/
     __u32     i_flags;                     /*打开文件的方式*/
     __u32     i_faddr;                     /*片的地址*/
     __u8 i_frag_no;                   /*如果用到片，则是第一个片号*/
     __u8 i_frag_size;                 /*片大小*/
      __u16    i_osync;                 /*同步*/
     __u32     i_file_acl;              /*文件访问控制链表*/
     __u32     i_dir_acl;                   /*目录访问控制链表*/
     __u32     i_dtime;                     /*文件的删除时间*/
     __u32     i_block_group;               /*索引节点所在的块组号*/
/******以下四个域是用于操作预分配块的*************/
     __u32     i_next_alloc_block;
     __u32     i_next_alloc_goal;
     __u32     i_prealloc_block;
     __u32     i_prealloc_count;

  __u32   i_dir_start_lookup
     int      i_new_inode:1   /* Is a freshly allocated inode */
};
```

VFS 索引节点中是没有物理块指针数组的域，这个 Ext2 特有的域在调入内存后，就必须保存在 ext2_inode_info 这个结构中。此外，片作为 Ext2 比较特殊的地方，在

ext2_inode_info 中也保存了一些相关的域。另外，Ext2 在分配一个块时通常还要预分配几个连续的块，因为它判断这些块很可能将要被访问，所以采用预分配的策略可以减少磁头的寻道时间。这些用于预分配操作的域也被保存在 ext2_inode_info 结构中。

9.2.4　组描述符

块组中，紧跟在超级块后面的是组描述符表，其每一项称为组描述符，是一个叫 ext2_group_desc 的数据结构，共 32 字节。它是用来描述某个块组的整体信息的。

```
struct ext2_group_desc
{
     __u32     bg_block_bitmap;        /*组中块位图所在的块号 */
     __u32     bg_inode_bitmap;             /*组中索引节点位图所在块的块号 */
     __u32     bg_inode_table;              /*组中索引节点表的首块号 */
     __u16     bg_free_blocks_count;   /*组中空闲块数 */
     __u16     bg_free_inodes_count;   /* 组中空闲索引节点数 */
     __u16     bg_used_dirs_count;          /*组中分配给目录的节点数 */
     __u16     bg_pad;                 /*填充，对齐到字*/
     __u32 [3]   bg_reserved;          /*用 NULL 填充 12 个字节*/
}
```

每个块组都有一个相应的组描述符来描述它，所有的组描述符形成一个组描述符表，组描述符表可能占多个数据块。组描述符就相当于每个块组的超级块，一旦某个组描述符遭到破坏，整个块组将无法使用，所以组描述符表也像超级块那样，在每个块组中进行备份，以防遭到破坏。组描述符表所占的块和普通的数据块一样，在使用时被调入块高速缓存。

9.2.5　位图

在 Ext2 中，是采用位图来描述数据块和索引节点的使用情况的，每个块组中都有两个块，一个用来描述该组中数据块的使用情况，另一个描述该组中索引节点的使用情况。这两个块分别称为数据位图块和索引节点位图块。数据位图块中的每一位表示该组中一个块的使用情况，如果为 0，则表示相应数据块空闲，为 1，则表示已分配，索引节点位图块的使用情况类似。

Ext2 在安装后，用两个高速缓存分别来管理这两种位图块。每个高速缓存最多同时只能装入 Ext2_MAX_GROUP_LOADED 个位图块或索引节点块，当前该值定义为 8，所以也应该采用一些算法来管理这两个高速缓存，Ext2 中采用的算法类似于 LRU 算法。

前面说过，ext2_sp_info 结构中有 4 个域用来管理这两个高速缓存，其中 s_block_bitmap_number[]数组中存有进入高速缓存的位图块号（即块组号，因为一个块组中只有一个位图块），而 s_block_bitmap[]数组则存储了相应的块在高速缓存中的地址。

s_inode_bitmap_number[]和 s_inode_bitmap[]数组的作用类似上面。

我们通过一个具体的函数来看 Ext2 是如何通过这 4 个域管理位图块管理高速缓存的。在 Linux/fs/ext2/balloc.c 中，有一个函数 load__block_bitmap()，它用来调入指定的数据位图块，下面是它的执行过程。

（1）如果指定的块组号大于块组数，出错，结束。

（2）通过搜索 s_block_bitmap_number[]数组可知位图块是否已进入高速缓存，如果已进入，则结束，否则，继续；

（3）如果块组数不大于 Ext2_MAX_GROUP_LOADED，高速缓存可以同时装入所有块组的数据块位图块，不用采用什么算法，只要找到 s_block_bitmap_number[]数组中第一个空闲元素，将块组号写入，然后将位图块调入高速缓存，最后将它在高速缓存中的地址写入 s_block_bitmap[]数组中。

（4）如果块组数大于 Ext2_MAX_GROUP_LOADED，则需要采用以下算法：

- 首先通过 s_block_bitmap_number[]数组判断高速缓存是否已满，若未满，则操作过程类似上一步，不同之处在于需要将 s_block_bitmap_number[]数组各元素依次后移一位，而用空出的第一个元素存储块组号，s_block_bitmap[]也要做相同处理；
- 如果高速缓存已满，则将 s_block_bitmap[]数组最后一项所指的位图块从高速缓存中交换出去，然后调入所指定的位图块，最后对这两个数组做与上面相同的操作。

可以看出，这个算法很简单，就是对两个数组的简单操作，只是在块组数大于 Ext2_MAX_GROUP_LOADED 时，要求数组的元素按最近访问的先后次序排列，显然，这样也是为了更合理的进行高速缓存的替换操作。

9.2.6 索引节点表及实例分析

在两个位图块后面，就是索引节点表了，每个块组中的索引节点都存储在各自的索引节点表中，并且按索引节点号依次存储。索引节点表通常占好几个数据块，索引节点表所占的块使用时也像普通的数据块一样被调入块高速缓存。

有了以上几个概念和数据结构后，我们分析一个具体的例子，来看看这些数据结构是如何配合工作的。

在 fs/ext2/inode.c 中，有一个 ext2_read_inode()，用来读取指定的索引节点信息。其代码如下：

```
void ext2_read_inode （struct inode * inode)
{
        struct buffer_head * bh;
        struct ext2_inode * raw_inode;
        unsigned long block_group;
        unsigned long group_desc;
        unsigned long desc;
        unsigned long block;
        unsigned long offset;
       struct ext2_group_desc * gdp;

        if （（inode->i_ino != EXT2_ROOT_INO && inode->i_ino != EXT2_ACL_IDX_INO &&
inode->i_ino != EXT2_ACL_DATA_INO && inode->i_ino < EXT2_FIRST_INO (inode->i_sb)） || inode->i_ino
> le32_to_cpu (inode->i_sb->u.ext2_sb.s_es->s_inodes_count) )
           {
               ext2_error （inode->i_sb, "ext2_read_inode",
                           "bad inode number: %lu", inode->i_ino) ;
```

```
        goto bad_inode;
}
block_group = (inode->i_ino - 1) / EXT2_INODES_PER_GROUP (inode->i_sb) ;
if (block_group >= inode->i_sb->u.ext2_sb.s_groups_count) {
        ext2_error (inode->i_sb, "ext2_read_inode",
                    "group >= groups count") ;
        goto bad_inode;
}
group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS (inode->i_sb) ;
desc = block_group & (EXT2_DESC_PER_BLOCK (inode->i_sb) - 1) ;
bh = inode->i_sb->u.ext2_sb.s_group_desc[group_desc];
if (!bh) {
        ext2_error (inode->i_sb, "ext2_read_inode",
                    "Descriptor not loaded") ;
        goto bad_inode;
}

gdp = (struct ext2_group_desc *) bh->b_data;
/*
 * Figure out the offset within the block group inode table
 */
offset = ( (inode->i_ino - 1) % EXT2_INODES_PER_GROUP (inode->i_sb) ) *
        EXT2_INODE_SIZE (inode->i_sb) ;
block = le32_to_cpu (gdp[desc].bg_inode_table) +
        (offset >> EXT2_BLOCK_SIZE_BITS (inode->i_sb) ) ;
if (! (bh = sb_bread (inode->i_sb, block) ) ) {
        ext2_error (inode->i_sb, "ext2_read_inode",
                    "unable to read inode block - "
                    "inode=%lu, block=%lu", inode->i_ino, block) ;
        goto bad_inode;
}
offset &= (EXT2_BLOCK_SIZE (inode->i_sb) - 1) ;
raw_inode = (struct ext2_inode *) (bh->b_data + offset) ;

inode->i_mode = le16_to_cpu (raw_inode->i_mode) ;
inode->i_uid = (uid_t) le16_to_cpu (raw_inode->i_uid_low) ;
inode->i_gid = (gid_t) le16_to_cpu (raw_inode->i_gid_low) ;
if (! (test_opt (inode->i_sb, NO_UID32) ) ) {
        inode->i_uid |= le16_to_cpu (raw_inode->i_uid_high) << 16;
        inode->i_gid |= le16_to_cpu (raw_inode->i_gid_high) << 16;
}
inode->i_nlink = le16_to_cpu (raw_inode->i_links_count) ;
inode->i_size = le32_to_cpu (raw_inode->i_size) ;
inode->i_atime = le32_to_cpu (raw_inode->i_atime) ;
inode->i_ctime = le32_to_cpu (raw_inode->i_ctime) ;
inode->i_mtime = le32_to_cpu (raw_inode->i_mtime) ;
inode->u.ext2_i.i_dtime = le32_to_cpu (raw_inode->i_dtime) ;
/* We now have enough fields to check if the inode was active or not.
 * This is needed because nfsd might try to access dead inodes
 * the test is that same one that e2fsck uses
 * NeilBrown 1999oct15
 */
```

```
        if (inode->i_nlink == 0 && (inode->i_mode == 0 || inode->u.ext2_i.i_dtime) ) {
                /* this inode is deleted */
                brelse (bh) ;
                goto bad_inode;
        }
        inode->i_blksize = PAGE_SIZE;   /* This is the optimal IO size (for stat) , not
the fs block size */
        inode->i_blocks = le32_to_cpu (raw_inode->i_blocks) ;
        inode->i_version = ++event;
        inode->u.ext2_i.i_flags = le32_to_cpu (raw_inode->i_flags) ;
        inode->u.ext2_i.i_faddr = le32_to_cpu (raw_inode->i_faddr) ;
        inode->u.ext2_i.i_frag_no = raw_inode->i_frag;
        inode->u.ext2_i.i_frag_size = raw_inode->i_fsize;
        inode->u.ext2_i.i_file_acl = le32_to_cpu (raw_inode->i_file_acl) ;
        if (S_ISREG (inode->i_mode) )
         inode->i_size |= ( (__u64) le32_to_cpu (raw_inode->i_size_high) ) <<32;
        else
              inode->u.ext2_i.i_dir_acl = le32_to_cpu (raw_inode->i_dir_acl) ;
        inode->i_generation = le32_to_cpu (raw_inode->i_generation) ;
        inode->u.ext2_i.i_prealloc_count = 0;
        inode->u.ext2_i.i_block_group = block_group;

        /*
         * NOTE! The in-memory inode i_data array is in little-endian order
         * even on big-endian machines: we do NOT byteswap the block numbers!
         */
        for (block = 0; block < EXT2_N_BLOCKS; block++)
                inode->u.ext2_i.i_data[block] = raw_inode->i_block[block];

        if (inode->i_ino == EXT2_ACL_IDX_INO ||
           inode->i_ino == EXT2_ACL_DATA_INO)
                /* Nothing to do */ ;
        else if (S_ISREG (inode->i_mode) ) {
                inode->i_op = &ext2_file_inode_operations;
                inode->i_fop = &ext2_file_operations;
                inode->i_mapping->a_ops = &ext2_aops;
        } else if (S_ISDIR (inode->i_mode) ) {
                inode->i_op = &ext2_dir_inode_operations;
                inode->i_fop = &ext2_dir_operations;
                inode->i_mapping->a_ops = &ext2_aops;
        } else if (S_ISLNK (inode->i_mode) ) {
                if (!inode->i_blocks)
                        inode->i_op = &ext2_fast_symlink_inode_operations;
                else {
                        inode->i_op = &page_symlink_inode_operations;
                        inode->i_mapping->a_ops = &ext2_aops;
                }
        } else
                init_special_inode (inode, inode->i_mode,
                                   le32_to_cpu (raw_inode->i_block[0]) ) ;
        brelse (bh) ;
        inode->i_attr_flags = 0;
```

```
        if (inode->u.ext2_i.i_flags & EXT2_SYNC_FL) {
                inode->i_attr_flags |= ATTR_FLAG_SYNCRONOUS;
                inode->i_flags |= S_SYNC;
        }
        if (inode->u.ext2_i.i_flags & EXT2_APPEND_FL) {
                inode->i_attr_flags |= ATTR_FLAG_APPEND;
                inode->i_flags |= S_APPEND;
        }
        if (inode->u.ext2_i.i_flags & EXT2_IMMUTABLE_FL) {
                inode->i_attr_flags |= ATTR_FLAG_IMMUTABLE;
                inode->i_flags |= S_IMMUTABLE;
        }
        if (inode->u.ext2_i.i_flags & EXT2_NOATIME_FL) {
                inode->i_attr_flags |= ATTR_FLAG_NOATIME;
                inode->i_flags |= S_NOATIME;
        }
        return;

bad_inode:
        make_bad_inode (inode) ;
        return;
}
```

这个函数的代码有 200 多行，为了突出重点，下面是对该函数主要内容的描述。

- 如果指定的索引节点号是一个特殊的节点号（EXT2_ROOT_INO、EXT2_ACL_IDX_ INO 及 EXT2_ACL_DATA_INO），或者小于第一个非特殊用途的节点号，即 EXT2_FIRST_ INO（为 11），或者大于该文件系统中索引节点总数，则输出错误信息，并返回。
- 用索引节点号整除每组中索引节点数，计算出该索引节点所在的块组号。

即：block_group = (inode->i_ino - 1) / Ext2_INODES_PER_GROUP(inode->i_sb)。

- 找到该组的组描述符在组描述符表中的位置。因为组描述符表可能占多个数据块，所以需要确定组描述符在组描述符表的哪一块以及是该块中第几个组描述符。

即：group_desc = block_group >> Ext2_DESC_PER_BLOCK_BITS(inode->i_sb) 表示块组号整除每块中组描述符数，计算出该组的组描述符在组描述符表中的哪一块。我们知道，每个组描述符是 32 字节大小，在一个 1KB 大小的块中可存储 32 个组描述符。

- 块组号与每块中组的描述符数进行“与”运算，得到这个组描述符具体是该块中第几个描述符。即 desc = block_group & (Ext2_DESC_PER_BLOCK(inode->i_sb) - 1)。
- 有了 group_desc 和 desc，接下来在高速缓存中找这个组描述符就比较容易了。

即：bh = inode->i_sb->u.ext2_sb.s_group_desc[group_desc]，首先通过 s_group_desc[]数组找到这个组描述符所在块在高速缓存中的缓冲区首部；然后通过缓冲区首部找到数据区，即 gdp = (struct ext2_group_desc *) bh->b_data。

- 找到组描述符后，就可以通过组描述符结构中的 bg_inode_tabl 找到索引节点表首块在高速缓存中的地址：

```
        offset = ((inode->i_ino - 1) % Ext2_INODES_PER_GROUP (inode->i_sb)) *
             Ext2_INODE_SIZE (inode->i_sb) /*计算该索引节点在块中的偏移位置*/;
block = le32_to_cpu (gdp[desc].bg_inode_table)   +
(offset >> Ext2_BLOCK_SIZE_BITS (inode->i_sb)) /*计算索引节点所在块的地址*/。
```

- 代码中 le32_to_cpu（）、le16_to_cpu（）按具体 CPU 的要求进行数据的排列，在 i386 处理器上访问 Ext2 文件系统时这些函数不做任何事情。因为不同的处理器在存取数据时在字节的排列次序上有所谓“big ending”和“little ending”之分。例如，i386 就是“little ending”处理器，它在存储一个 16 位数据 0x1234 时，实际存储的却是 0x3412，对 32 位数据也是如此。这里索引节点号与块的长度都作为 32 位或 16 位无符号整数存储在磁盘上，而同一磁盘既可以安装在采用“little ending”方式的 CPU 机器上，也可能安装在采用“big ending”方式的 CPU 机器上，所以要选择一种形式作为标准。事实上，Ext2 采用的标准为“little ending”，所以，le32_to_cpu（）、le16_to_cpu（）函数不作任何转换。
- 计算出索引节点所在块的地址后，就可以调用 sb_bread（）通过设备驱动程序读入该块。从磁盘读入的索引节点为 ext2_Inode 数据结构，前面我们已经看到它的定义。磁盘上索引节点中的信息是原始的、未经加工的，所以代码中称之为 raw_ inode，即：raw_inode = (struct ext2_inode *) (bh->b_data + offset)
- 与磁盘索引节点 ext2_ inode 相对照，内存中 VFS 的 inode 结构中的信息则分为两部分，一部分是属于 VFS 层的，适用于所有的文件系统；另一部分则属于具体的文件系统，这就是 inode 中的那个 union，因具体文件系统的不同而赋予不同的解释。对 Ext2 来说，这部分数据就是前面介绍的 ext2_inode_info 结构。至于代表着符号链接的节点，则并没有文件内容（数据），所以正好用这块空间来存储链接目标的路径名。ext2_inode_info 结构的大小为 60 个字节。虽然节点名最长可达 255 个字节，但一般都不会太长，因此将符号链接目标的路径名限制在 60 个字节不至于引起问题。代码中 inode->u.*设置的就是 Ext2 文件系统的特定信息。
- 接着，根据索引节点所提供的信息设置 inode 结构中的 inode_operations 结构指针和 file_operations 结构指针，完成具体文件系统与虚拟文件系统 VFS 之间的连接。
- 目前 2.4 版内核并不支持存取控制表 ACL，因此，代码中只是为之留下了位置，而暂时没做任何处理。
- 另外，通过检查 inode 结构中的 mode 域来确定该索引节点是常规文件（S_ISREG）、目录（S_ISDIR）、符号链接（S_ISLNK）还是其他特殊文件而作不同的设置或处理。例如，对 Ext2 文件系统的目录节点，就将 i_op 和 i_fop 分配设置为 ext2_dir_inode_operations 和 ext2_dir_operations。而对于 Ext2 常规文件，则除 i_op 和 i_fop 以外，还设置了另一个指针 a_ops，它指向一个 address_apace_operation 结构，用于文件到内存空间的映射或缓冲。对特殊文件，则通过 init_special_inode（）函数加以检查和处理。

从这个读索引节点的过程可以看出，首先要寻找指定的索引节点 ，要找索引节点，必须先找组描述符，然后通过组描述符找到索引节点表，最后才是在这个索引节点表中找索引节点。当从磁盘找到索引节点以后，就要把其读入内存，并存放在 VFS 索引节点相关的域中。从这个实例的分析，读者可以仔细体会前面所介绍的各种数据结构的具体应用。

9.2.7　Ext2 的目录项及文件的定位

文件系统一个很重要的问题就是文件的定位，如何通过一个路径来找到一个文件的具体位置，就要依靠 ext2_dir_entry 这个结构。

1. Ext2 目录项结构

在 Ext2 中，目录是一种特殊的文件，它是由 ext2_dir_entry 这个结构组成的列表。这个结构是变长的，这样可以减少磁盘空间的浪费，但是，它还是有一定的长度方面的限制，一是文件名最长只能为 255 个字符。二是尽管文件名长度可以不限（在 255 个字符之内），但系统自动将之变成 4 的整数倍，不足的地方用 NULL 字符（\0）填充。目录中有文件和子目录，每一项对应一个 ext2_dir_entry。该结构在 include/Linux/ext2_fs.h 中定义如下：

```
 /*
  * Structure of a directory entry
  */
  #define EXT2_NAME_LEN 255
struct ext2_dir_entry {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u16   name_len;               /* Name length */
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

这是老版本的定义方式，在 ext2_fs.h 中还有一种新的定义方式：

```
 /*
  * The new version of the directory entry.  Since EXT2 structures are
  * stored in intel byte order, and the name_len field could never be
  * bigger than 255 chars, it's safe to reclaim the extra byte for the
  * file_type field.
  */
struct ext2_dir_entry_2 {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
       __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

其二者的差异在于，一是新版中结构名改为 ext2_dir_entry_2；二是老版本中 ext2_dir_entry 中的 name_len 为无符号短整数，而新版中则改为 8 位的无符号字符，腾出一半用作文件类型。目前已定义的文件类型为：

```
  *
  * Ext2 directory file types.  Only the low 3 bits are used.  The
  * other bits are reserved for now.
  */
enum {
        EXT2_FT_UNKNOWN,   /*未知*/
        EXT2_FT_REG_FILE,  /*常规文件*/
        EXT2_FT_DIR,            /*目录文件*/
        EXT2_FT_CHRDEV,    /*字符设备文件*/
        EXT2_FT_BLKDEV,    /*块设备文件*/
        EXT2_FT_FIFO,      /*命名管道文件*/
        EXT2_FT_SOCK,      /*套接字文件*/
        EXT2_FT_SYMLINK,   /*符号连文件*/
        EXT2_FT_MAX             /*文件类型的最大个数*/
```

```
};
```

2．各种文件类型如何使用数据块

我们说，不管哪种类型的文件，每个文件都对应一个 inode 结构，在 inode 结构中有一个指向数据块的指针 i_blaock，用来标识分配给文件的数据块。但是 Ext2 所定义的文件类型以不同的方式使用数据块。有些类型的文件不存放数据，因此，根本不需要数据块，下面对不同文件类型如何使用数据块给予说明。

（1）常规文件

常规文件是最常用的文件。常规文件在刚创建时是空的，并不需要数据块，只有在开始有数据时才需要数据块；可以用系统调用 truncate()清空一个常规文件。

（2）目录文件

Ext2 以一种特殊的方式实现了目录,这种文件的数据块中存放的就是 ext2_dir_entry_2 结构。如前所述，这个结构的最后一个域是可变长度数组，因此该结构的长度是可变的。

在 ext2_dir_entry_2 结构中，因为 rec_len 域是目录项的长度，把它与目录项的起始地址相加就得到下一个目录项的起始地址，因此说，rec_len 可以被解释为指向下一个有效目录项的指针。为了删除一个目录项，把 ext2_dir_entry_2 的 inode 域置为 0 并适当增加前一个有效目录项 rec_len 域的值就可以了。

（3）符号连

如果符号连的路径名小于 60 个字符，就把它存放在索引节点的 i_blaock 域，该域是由 15 个 4 字节整数组成的数组，因此无需数据块。但是，如果路径名大于 60 个字符，就需要一个单独的数据块。

（4）设备文件、管道和套接字

这些类型的文件不需要数据块。所有必要的信息都存放在索引节点中。

3．文件的定位

文件的定位是一个复杂的过程，我们先看一个具体的例子，然后再结合上面的数据结构具体介绍一下如何找到一个目录项的过程。

如果要找的文件为普通文件，则可通过文件所对应的索引节点找到文件的具体位置，如果是一个目录文件，则也可通过相应的索引节点找到目录文件具体所在，然后再从这个目录文件中进行下一步查找，来看一个具体的例子。

假设路径为/home/user1/file1，home 和 user1 是目录名，而 file1 为文件名。为了找到这个文件，有两种途径，一是从根目录开始查找，二是从当前目录开始查找。假设我们从根目录查找，则必须先找到根目录的节点，这个节点的位置在 VFS 中的超级块中已经给出，然后可找到根目录文件，其中必有 home 所对应的目录项，由此可先找到 home 节点，从而找到 home 的目录文件，然后依次是 user1 的节点和目录文件，最后，在该目录文件中的 file1 目录项中找到 file1 的节点，至此，已经可以找到 file1 文件的具体所在了。

目录中还有两个特殊的子目录：“．”和“．．”，分别代表当前目录和父目录。它们是无法被删除的，其作用就是用来进行相对路径的查找。

现在，我们来分析一下 fs/ext2/dir.c 中的函数 ext2_find_entry（），该函数从磁盘

上找到并读入当前节点的目录项，其代码及解释如下：

```
/*
 *      ext2_find_entry ()
 *
 * finds an entry in the specified directory with the wanted name. It
 * returns the page in which the entry was found, and the entry itself
 *  (as a parameter - res_dir) . Page is returned mapped and unlocked.
 * Entry is guaranteed to be valid.
 */
typedef struct ext2_dir_entry_2 ext2_dirent;
struct ext2_dir_entry_2 * ext2_find_entry (struct inode * dir,
                        struct dentry *dentry, struct page ** res_page)
{
      const char *name = dentry->d_name.name;       ／*目录项名*/
      int namelen = dentry->d_name.len;                  ／*目录项名的长度*/
      unsigned reclen = EXT2_DIR_REC_LEN (namelen) ;     ／*目录项的长度*/
      unsigned long start, n;
      unsigned long npages = dir_pages (dir) ;           ／*把以字节为单位的文件大小
转换为物理页面数*/
        struct page *page = NULL;
         ext2_dirent * de;                                    ／*de 为要返回的 Ext2 目录项
                                                              ／*结构*/
         /* OFFSET_CACHE */
         *res_page = NULL;

        start = dir->u.ext2_i.i_dir_start_lookup;   ／*目录项在内存的起始位置*/
         if  (start >= npages)
                start = 0;
         n = start;
        do {
                char *kaddr;
                page = ext2_get_page (dir, n) ;         ／*从页面高速缓存中获得目录
项所在的页面*／
                if  (!IS_ERR (page) )  {
                     kaddr = page_address (page) ;     ／*获得 page 所对应的内核
虚拟地址*／
                        de =  (ext2_dirent *)  kaddr;   ／*获得该目录项结构的
起始地址*/
                         kaddr += PAGE_CACHE_SIZE - reclen; ／* PAGE_CACHE_SIZE
的大小为 1 个页面的大小，假定所有的目录项结构都存放在一个页面内*/
                        while  ( (char *)  de <= kaddr)  {  ／*循环查找，直到
  找到匹配的目录项*/
                                if  (ext2_match  (namelen, name, de) )
                                      goto found;
                              de = ext2_next_entry (de) ;
                        }
                       ext2_put_page (page) ;      /*释放目录项所在的页面*／
               }
               if  (++n >= npages)
                        n = 0;
        } while  (n != start) ;
         return NULL;
```

```
found:
        *res_page = page;
        dir->u.ext2_i.i_dir_start_lookup = n;
        return de;
}
```

通过代码的注释,读者应当很容易明理解其含义。

9.3 文件的访问权限和安全

Linux 作为一种网络操作系统，允许多个用户使用。为了保护用户的个人文件不被其他用户侵犯，Linux 提供了文件权限的机制。这种机制使得一个文件或目录归一个特定的用户所有，这个用户有权对他所拥有的文件或目录进行存取或其他操作，也可以设置其他用户对这些文件或目录的操作权限。

Linux 还用到了用户组的概念。每个用户在建立用户目录时都被放到至少一个用户组中（当然，系统管理员可以将用户编进多个用户组中）。用户组通常是根据使用计算机的用户的种类来划分的。例如，普通用户通常属于 usrs 组。另外，还有几个系统定义的组（如 bin 和 admin），系统使用这些组来控制对资源的访问。

普通文件的权限有 3 部分：读、写和执行。分别用“r”、“w”、“x”来表示。目录也有这 3 种表示方式，分别表示：列出目录内容、在目录中建立或删除文件、进入和退出目录。以一个例子来解释文件权限的概念。

我们用带-l 选项的 ls 命令来显示包括文件权限的长格式信息。

```
/ $  ls  -l   /home/user1/file1
-rw-r--r--     1      user1           usrs       490      JUN  28  21:50     file1
```

在这个文件的长格式信息中，第 1 列代表的就是文件的类型和权限。其中，第 1 个字母表示文件的类型，“-”表明这是一个普通文件。接下来 9 个字符每 3 个一组依次代表文件的所有者、所有者所在用户组和组外其他用户对该文件的访问权限，代表文件权限的 3 个字符依次是读、写和执行权限，当用户没有相应的权限时，系统在该权限对应的位置上用“-”表示。所以，本例表示用户 user1 自己对该文件有读和写的权限，但没有执行权限，而 user1 所在 usrs 组的其他用户和组外的用户对该文件只有读权限，没有写和执行权限。

有时候，我们会看到权限的执行位上出现了“s”，而不是“x”。这涉及到进程中的概念。一个进程，除了进程标识号（PID）外，还有 4 个标识号表示进程的权限。它们是：

- real user ID　　实际用户标识号
- real group ID　　实际用户组标识号
- effective user ID　　有效用户标识号
- effective group ID　　有效用户组标识号

实际用户标识号就是运行该进程的用户的 UID，实际用户组标识号就是运行该进程的用户的 GID。一般情况下，有效用户标识号、有效用户组标识号分别和实际用户标识号、实际用户组标识号相同。但如果设置 setuid 位和 setgid 位，则这两个标识号在对文件进行操作时自动转换成该文件所有者和所有组的标识号。setuid 位和 setgid 位设置与否，就是看文

件权限位上是否有“s”，如果用户的执行权限位为“s”，则设置了 setuid 位，如果用户组的执行权限位为“s”，则设置了 setgid 位。

这两个标识位的设置正确与否关系到整个系统的安全，所以非同小可，我们以两个例子来说明。

/bin/passwd 就是一个设置了 setuid 位的文件。当/bin/passwd 被执行时，它要在/etc 目录下找一个也叫 passwd 的文件，该文件是个文本文件，里面记录了全部用户的数据，如口令、用户标识号等。该文件显然不能让超级用户以外的人修改，所以该文件的存取权限是 rw-r--r--。但是，普通用户在更改它自己的口令时，必须改动这个文件，解决这个问题的方法便是设置 setuid 的位，使指令在被执行时有和 root 相同的权利，这样就可以修改这个文件了。

setuid 位的设立是有风险的！如果某个普通进程执行一个设置了 setuid 位，且所有者是“root”的文件，立刻具有了超级用户的权利，万一该程序有 BUG，就留下了被人入侵系统的可能。因此，把文件按性质分类，再用设置 setgid 位来达到分享的目标是一个不错的办法。

Linux 中有一个叫/etc/kmem 的字符设备文件，目的为存储一些核心程序要用到的数据。当用户注册到系统时所输入的口令都存在其中，所以这个文件不能给普通用户读写，以免给企图侵入系统者可乘之机，但将使用权限设为“rw-------”也不行，这样一般用户无法用 ps 等显示系统状态的指令（这类显示系统状态的指令必须要读取 kmem 的内容）。如果将存取权限设置成“rw-r-----”，再把组标识号改成与 kmem 一样，最后再设置 setgid 位，这样，无论谁执行 ps，该进程的组标识号都变得和 ps 一样，从而就能读取 kmem 的数据了。

作为一个系统管理员，能用 setgid 来代替 setuid 就尽量用 setgid，因为它的风险小得多！

还有一个不引人注意的现象，就是在其他用户的执行权限位上可能出现“t”，而不是“x”，这表示该文件被装入内存后，将一直保留在内存中，就像 dos 下 TSR 程序，这样可以减少程序的装入时间，增加程序的反应速度。因此，系统管理员可以对一些使用频率较高的工具程序设置该位。

创建一个新文件时，需要提供文件的访问权限位，但是，新文件的权限位并不就是根据这个数设置的，还必须参考另一个值：文件权限的掩码，这是一个 9 位的二进制数，对应于 9 个权限位，如果这个数某一位上置 1，则新创建文件相应权限位被强制置 0，而不管所提供的权限位是怎样的。这种机制使的一个新文件在创建时被限制为一个合适的权限。可以用 umask 命令看到这个值，它以 3 位八进制数显示，默认为 022，即 000010010，所以一个新文件创建时，除了文件拥有者外，其他用户是没有写权限的。在 VFS 的 fs_struct 结构中有一个 umask 域，存储的就是这个文件权限的掩码。

文件建立后，文件拥有者可以用 chmod 改变访问权限。每一组权限位的设置可以用一个八进制数表示，这样，用 3 个八进制数就可以表示 9 个权限位了。例如 chmod 755/home/user1/file1，则该文件的权限表示为 wxr-xr-xr。如果要设置 setuid、setgid 位，则需要用到第 4 个八进制数，该数为 4，则 setuid 位被设置；该数为 2，则 setgid 位被设置，该数为 1，则是其他用户的执行位被设置成“t”。

Ext2 索引节点中的 i_mode 就是用来存储文件的属性、用户标识号、用户组标识号和访

问权限等信息的，这是一个 16 位的字段，其中 0 到 8 位用来表示文件的访问权限，第 9 位用来控制文件是否驻留内存，10、11 位即 setgid、setuid 位，12 到 15 位的组合用来表示文件类型。如表 9.1 所示。

表 9.1　　　　imode 字段各位的含义

<table>
<tr><th colspan="2">位</th><th>符号常量</th><th>含义</th></tr>
<tr><td rowspan="7">12~15 位
的组合</td><td>1100</td><td>S_IFSOCK</td><td>套接字</td></tr>
<tr><td>1010</td><td>S_IFLNK</td><td>符号链接文件</td></tr>
<tr><td>1000</td><td>S_IFREG</td><td>普通文件</td></tr>
<tr><td>0110</td><td>S_IFBLK</td><td>块设备文件</td></tr>
<tr><td>0100</td><td>S_IFDIR</td><td>目录</td></tr>
<tr><td>0010</td><td>S_IFCHR</td><td>字符设备文件</td></tr>
<tr><td>0001</td><td>S_IFIFO</td><td>命名管道</td></tr>
<tr><td colspan="2">11</td><td>S_ISUID</td><td>用户</td></tr>
<tr><td colspan="2">10</td><td>S_ISGID</td><td>用户组</td></tr>
<tr><td colspan="2">9</td><td>S_ISVTX</td><td>文件是否驻留内存</td></tr>
<tr><td colspan="2">0~8</td><td>rwx-rwx-rwx</td><td>文件的访问权限</td></tr>
</table>

这个表中的符号常量都是在 include/Linux/stat.h 中定义的。

9.4　链接文件

由前面有关索引节点和目录项的介绍，我们已经知道 Ext2 把文件名和文件信息分开存储，其中文件信息用索引节点来描述，目录项就是用来联系文件名和索引节点的。目录项中，每一对文件名和索引节点号的一个一一对应称为一个链接，这就使同一个索引节点号出现在多个链接中成为可能，也就是说，同一个索引节点号可以对应多个不同的文件名。这种链接称为硬链接，可以用 ln 命令为一个已存在的文件建立一个新的硬链接：

```
ln   /home/user1/file1    /home/user1/file2
```

建立了一个文件 file2，链接到 file1 上。file2 和 file1 有相同的索引节点号，也就是和 file1 共享同一个索引节点。在建立了一个新的硬链接后，这个索引节点中的 i_links_count 值将加 1，i_links_count 的值反映了链接到这个索引节点上的文件数。

使用硬链接的好处如下所示。

（1）由于在删除文件时，实际上先对 i_links_count 作减 1，如果 i_links_count 不为 0，则结束，即仅仅删除了一个硬链接，具体文件的数据并没有被删除。只有在 i_links_count 为 0 时，才真正将文件从磁盘上删除。这样，你可以对重要的文件作多个链接，防止文件被误

删除。

(2) 允许用户在不进入某个目录的情况下对该目录下面的文件进行处理。

由于同一个文件系统中，索引节点号是系统用来辨认文件的唯一标志，而两个不同的文件系统中，可能有索引节点号一样的文件，所以硬链接仅允许在同一个文件系统上进行，要在多个文件系统之间建立链接，必须用到符号链接。

符号链接与硬链接最大的不同就在于它并不与索引节点建立链接，也就是说当为一个文件建立一个符号链接时，索引节点的链接计数并不变化。当你删除一个文件时，它的符号链接文件也就失去了作用，而当你删去一个文件的符号链接文件，对该文件本身并无影响。所以，有必要区分符号链接文件和硬链接文件，符号链接文件用“ l ”表示，另外，符号链接文件的索引节点号与原文件的索引节点号也是不同的。而硬链接只是普通文件。硬链接的一个缺陷是你无法简单地知道哪些文件是链接到同一个文件上的。而在符号链接中，可以看到它是指向哪个文件的。

最后，硬链接只能由超级用户建立，而普通用户可以建立符号链接。建立符号链接用 ls -s 命令。

因为内核为符号链接文件也创建一个索引节点，但它跟普通文件的索引节点所有不同。如前所述，代表着链接节点的文件没有数据，因此，关于符号链接的操作也就比较简单。对 Ext2 文件系统来说，只有 ext2_readlink() 和 ext2_follow_link() 函数，这是在 fs/ext2/symlink.c 中定义的：

```
struct inode_operations ext2_fast_symlink_inode_operations = {
        readlink:       ext2_readlink,
        follow_link:    ext2_follow_link,
};
```

ext2_readlink()函数的代码如下：

```
static int ext2_readlink (struct dentry *dentry, char *buffer, int buflen)
{
        char *s =  (char *) dentry->d_inode->u.ext2_i.i_data;
        return vfs_readlink (dentry, buffer, buflen, s) ;
}
```

如前所述，对于 Ext2 文件系统，连接目标的路径在 ext2_ inode_info 结构（即 inode 结构的 union 域）的 i_data 域中存放，因此字符串 s 就存放有连接目标的路径名。vfs_readlink（）的代码在 fs/namei.c 中：

```
int vfs_readlink (struct dentry *dentry, char *buffer, int buflen, const char *link)
{
        int len;

        len = PTR_ERR (link) ;
        if  (IS_ERR (link) )
                goto out;

        len = strlen (link) ;
        if  (len >  (unsigned)  buflen)
                len = buflen;
        if  (copy_to_user (buffer, link, len) )
                len = -EFAULT;
out:
```

```
        return len;
}
```

从代码可以看出，该函数比较简单，即把连接目标的路径名拷贝到用户空间的缓冲区中，并返回路径名的长度。

ext2_follow_link（）函数用于搜索符号连接所在的目标文件，其代码如下：

```
static int ext2_follow_link (struct dentry *dentry, struct nameidata *nd)
{
        char *s =  (char *) dentry->d_inode->u.ext2_i.i_data;
        return vfs_follow_link (nd, s) ;
}
```

这个函数与 ext2_readlink（）类似，值得注意的是，从 ext2_readlink（）中对 vfs_readlink（）的调用意味着从较低的层次（Ext2 文件系统）回到更高的 VFS 层次。为什么呢？这是因为符号链接的目标有可能在另一个不同的文件系统中，因此，必须通过 VFS 来中转，在 vfs_follow_link（）中必须要调用路径搜索函数 link_path_walk()来找到代表着连接对象的 dentry 结构，函数的代码如下：

```
static inline int vfs_follow_link (struct nameidata *nd, const char *link)
{
        int res = 0;
        char *name;
        if  (IS_ERR (link) )
                goto fail;

        if  (*link == '/')  {
                path_release (nd) ;
                if  (!walk_init_root (link, nd) )
                        /* weird __emul_prefix ()  stuff did it */
                        goto out;
        }
        res = link_path_walk (link, nd) ;
out:
        if  (current->link_count || res || nd->last_type!=LAST_NORM)
                return res;
        /*
         * If it is an iterative symlinks resolution in open_namei ()  we
         * have to copy the last component. And all that crap because of
         * bloody create ()  on broken symlinks. Furrfu...
         */
        name = __getname () ;
        if  (!name)
                return -ENOMEM;
        strcpy (name, nd->last.name) ;
        nd->last.name = name;
        return 0;
fail:
        path_release (nd) ;
        return PTR_ERR (link) ;
}
```

其中 nameidata 结构为：

```
struct nameidata {
```

```
        struct dentry *dentry;
        struct vfsmount *mnt;
         struct qstr last;
        unsigned int flags;
         int last_type;
};
```

last_type 域的可能取值定义于 fs.h 中：

```
/*
* Type of the last component on LOOKUP_PARENT
*/
enum {LAST_NORM, LAST_ROOT, LAST_DOT, LAST_DOTDOT, LAST_BIND};
```

在路径的搜索过程中，这个域的值会随着路径名当前的搜索结果而变。例如，如果成功地找到了目标文件，那么这个域的值就变成了 LAST_NORM；而如果最后停留在一个“ . ”上，则变成 LAST_DOT。

Qstr 结构用来存放路径名中当前节点的名字、长度及哈希值，其定义于 include/linux/dcache.h 中：

```
*
* "quick string" -- eases parameter passing, but more importantly
* saves "metadata" about the string  (ie length and the hash) .
*/

struct qstr {
        const unsigned char * name;
        unsigned int len;
        unsigned int hash;
};
```

下面来对 vfs_follow_link（）函数的代码给予说明。

- 如果符号链接的路径名是以“ ／ ”开头的绝对路径，那就要通过 walk_init_root（）从根节点开始查找。
- 调用 link_path_walk（）函数查找符号链接所在目标文件对应的信息。从 link_path_walk()返回时,返回值为 0 表示搜索成功,此时,nameidata 结构中的指针 dentry 指向目标节点的 dentry 结构，指针 mnt 指向目标节点所在设备的安装结构，同时，这个结构中的 last_type 表示最后一个节点的类型，节点名则在类型为 qstr 结构的 last 中。该函数失败时，则函数返回值为一负的出错码，而 nameidata 结构中则提供失败的节点名等信息。
- vfs_follow_link（）返回值的含义与 ink_path_walk（）函数完全相同。

9.5　分配策略

当建立一个新文件或目录时，Ext2 必须决定在磁盘上的什么地方存储数据，也就是说，将哪些物理块分配给这个新文件或目录。一个好的分配物理块的策略，将导致文件系统性能的提高。一个好的思路是将相关的数据尽量存储在磁盘上相邻的区域，以减少磁头的寻道时间。Ext2 使用块组的优越性就体现出来了，因为，同一个组中的逻辑块所对应的物理块通常是相邻存储的。Ext2 企图将每一个新的目录分到它的父目录所在的组，因为在理论上，访问

完父目录后，接着要访问其子目录，例如对一个路径的解析。所以，将父目录和子目录放在同一个组是有必要的。它还企图将文件和它的目录项分在同一个组，因为目录访问常常导致文件访问。当然如果组已满，则文件或目录可能分在某一个未满的组中。

分配新块的算法如下所述。

（1）文件的数据块尽量和它的索引节点在同一个组中。

（2）每个文件的数据块尽量连续分配。

（3）父目录和子目录尽量在一个块组中。

（4）文件和它的目录项尽量在同一个块组中。

9.5.1 数据块寻址

每个非空的普通文件都是由一组数据块组成。这些块或者由文件内的相对位置（文件块号）来表示，或者由磁盘分区内的位置（它们的逻辑块号）来表示。

从文件内的偏移量 f 导出相应数据块的逻辑块号需要以下两个步骤。

- 从偏移量 f 导出文件的块号，即偏移量 f 处的字符所在的块索引。
- 把文件的块号转化为相应的逻辑块号。

因为 Linux 文件不包含任何控制字符，因此，导出文件的第 f 个字符所在的文件块号是相当容易的：只是用 f 除以文件系统块的大小，并取整即可。

例如，让我们假定块的大小为 4KB。如果 f 小于 4096，那么这个字符就在文件的第 1 个数据块中，其文件的块号为 0。如果 f 等于或大于 4096 而小于 8192，则这个字符就在文件块号为 1 的数据块中等等。

只用关心文件的块号确实不错。但是，由于 Ext2 文件的数据块在磁盘上并不是相邻的，因此把文件的块号转化为相应的逻辑块号可不是那么直接了当。

因此，Ext2 文件系统必须提供一种方法，用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。在索引节点内部部分实现了这种映射，这种映射也包括一些专门的数据块，可以把这些数据块看成是用来处理大型文件的索引节点的扩展。

磁盘索引节点的 i_block 域是一个有 EXT2_N_BLOCKS 个元素且包含逻辑块号的数组。在下面的讨论中，我们假定 EXT2_N_BLOCKS 的默认值为 15，如图 9.4 所示，这个数组表示一个大型数据结构的初始化部分。正如你从图中所看到的，数组的 15 个元素有 4 种不同的类型。

- 最初的 12 个元素产生的逻辑块号与文件最初的 12 个块对应，即对应的文件块号从 0 到 11。
- 索引 12 中的元素包含一个块的逻辑块号，这个块代表逻辑块号的一个二级数组。这个数组对应的文件块号从 12 到 $b/4+11$，这里 b 是文件系统的块大小（每个逻辑块号占 4 个字节，因此我们在式子中用 4 做除数）。因此，内核必须先用指向一个块的指针访问这个元素，然后，用另一个指向包含文件最终内容的块的指针访问那个块。
- 索引 13 中的元素包含一个块的逻辑块号，这个块包含逻辑块号的一个二级数组；这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是逻辑块号对应的文件块号，范围从 $b/4+12$ 到 $(b/4)^2+(b/4)+11$。
- 最后，索引 14 中的元素利用了三级间接索引：第四级数组中存放的才是逻辑块号对

应的文件块号，范围从 $(b/4)^2+(b/4)+12$ 到 $(b/4)^3+(b/4)^2+(b/4)+11$。

注意这种机制是如何支持小文件的。如果文件需要的数据块小于 12，那么两次访问磁盘就可以检索到任何数据：一次是读磁盘索引节点 i_block 数组的一个元素，另一次是读所需要的数据块。对于大文件来说，可能需要 3～4 次的磁盘访问才能找到需要的块。实际上，这是一种最坏的估计，因为目录项、缓冲区及页高速缓存都有助于极大地减少实际访问磁盘的次数。

也要注意文件系统的块大小是如何影响寻址机制的，因为大的块大小允许 Ext2 把更多的逻辑块号存放在一个单独的块中。表 9.2 显示了对每种块大小和每种寻址方式所存放文件大小的上限。例如，如果块的大小是 1024 字节，并且文件包含的数据最多为 268KB，那么，通过直接映射可以访问文件最初的 12KB 数据，通过简单的间接映射可以访问剩余的 13KB 到 268KB 的数据。对于 4096 字节的块，两次间接就完全满足了对 2GB 文件的寻址（2GB 是 32 位体系结构上的 Ext2 文件系统所允许的最大值）。

表 9.2　　可寻址的文件数据块大小的界限

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	63.55 MB	2 GB
2048	24 KB	1.02 MB	513.02 MB	2 GB
4096	48 KB	4.04 MB	2 GB	—

9.5.2　文件的洞

文件的洞是普通文件的一部分，它是一些空字符但没有存放在磁盘的任何数据块中。洞是 UNIX 文件一直存在的一个特点。例如，下列的 Linux 命令创建了第一个字节是洞的文件。

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，/tmp/hole 有 6145 个字符（6144 个 NULL 字符加一个 X 字符），然而，这个文件只占磁盘上一个数据块。

引入文件的洞是为了避免磁盘空间的浪费。它们被广泛地用在数据库应用中，更一般地说，用于在文件上执行散列法的所有应用。

文件洞在 Ext2 的实现是基于动态数据块的分配：只有当进程需要向一个块写数据时，才真正把这个块分配给文件。每个索引节点的 i_size 域定义程序所看到的文件大小，包括洞，而 i_blocks 域存放分配给文件有效的数据块数（以 512 字节为单位）。

在前面 dd 命令的例子中，假定/tmp/hole 文件被创建在块大小为 4096 的 Ext2 分区上。其相应磁盘索引节点的 i_size 域存放的数为 6145，而 i_blocks 域存放的数为 8（因为每 4096 字节的块包含 8 个 512 字节的块）。i_block 数组的第 2 个元素（对应块的文件块号为 1）存放已分配块的逻辑块号，而数组中的其他元素都为空。

9.5.3　分配一个数据块

当内核要分配一个新的数据块来保存 Ext2 普通文件的数据时，就调用

ext2_get_block( )函数。这个函数依次处理在“数据块寻址”部分所描述的那些数据结构，并在必要时调用 ext2_alloc_block( )函数在 Ext2 分区实际搜索一个空闲的块。

为了减少文件的碎片，Ext2 文件系统尽力在已分配给文件的最后一个块附近找一个新块分配给该文件。如果失败，Ext2 文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。作为最后一个办法，可以从其他一个块组中获得空闲块。

Ext2 文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块，而是获得一组多达 8 个邻接的块。ext2_inode_info 结构的 i_prealloc_count 域存放预分配给某一文件但还没有使用的数据块数，而 i_prealloc_block 域存放下一次要使用的预分配块的逻辑块号。当下列情况发生时，即文件被关闭时，文件被删除时，或关于引发块预分配的写操作而言，有一个写操作不是顺序的时候，就释放预分配但一直没有使用的块。

下面我们来看一下 ext2_get_block( )函数，其代码在 fs/ext2/inode.c 中：

```
  /*
   * Allocation strategy is simple: if we have to allocate something, we will
   * have to go the whole way to leaf. So let's do it before attaching anything
   * to tree, set linkage between the newborn blocks, write them if sync is
   * required, recheck the path, free and repeat if check fails, otherwise
   * set the last missing link (that will protect us from any truncate-generated
   * removals - all blocks on the path are immune now) and possibly force the
   * write on the parent block.
   * That has a nice additional property: no special recovery from the failed
   * allocations is needed - we simply release blocks and do not touch anything
   * reachable from inode.
   */
static int ext2_get_block (struct inode *inode, long iblock, struct buffer_head *bh_result,
int create)
  {
        int err = -EIO;
        int offsets[4];
        Indirect chain[4];
        Indirect *partial;
        unsigned long goal;
        int left;
        int depth = ext2_block_to_path (inode, iblock, offsets) ;
        if  (depth == 0)
              goto out;

       lock_kernel () ;
reread:
        partial = ext2_get_branch (inode, depth, offsets, chain, &err) ;

        /* Simplest case - block found, no allocation needed */
        if  (!partial)  {
got_it:
              bh_result->b_dev = inode->i_dev;
              bh_result->b_blocknr = le32_to_cpu (chain[depth-1].key) ;
              bh_result->b_state |=  (1UL << BH_Mapped) ;
               /* Clean up and exit */
               partial = chain+depth-1; /* the whole chain */
```

```
                goto cleanup;
        }

        /* Next simple case - plain lookup or failed read of indirect block */
        if (!create || err == -EIO) {
cleanup:
                while (partial > chain) {
                        brelse (partial->bh) ;
                        partial--;
                }
                unlock_kernel () ;
out:
                return err;
        }

        /*
         * Indirect block might be removed by truncate while we were
         * reading it. Handling of that case (forget what we've got and
         * reread) is taken out of the main path.
         */
        if (err == -EAGAIN)
                goto changed;

        if (ext2_find_goal (inode, iblock, chain, partial, &goal) < 0)
                goto changed;

        left = (chain + depth) - partial;
        err = ext2_alloc_branch (inode, left, goal,
                                        offsets+ (partial-chain) , partial) ;
        if (err)
                goto cleanup;

        if (ext2_splice_branch (inode, iblock, chain, partial, left) < 0)
                goto changed;

        bh_result->b_state |= (1UL << BH_New) ;
        goto got_it;

changed:
        while (partial > chain) {
                brelse (partial->bh) ;
                partial--;
        }
        goto reread;
}
```

对这个函数，源代码的作者对分配策略给出了简单的注释，相信读者从会中领略一二。函数的参数 inode 指向文件的 inode 结构；参数 iblock 表示文件中的逻辑块号；参数 bh_result 为指向缓冲区首部的指针，buffer_head 结构已在上一章做了介绍；参数 create 表示是否需要创建。

其中 Indirect 结构在同一文件中定义如下：

```
typedef struct {
        u32     *p;
        u32     key;
        struct buffer_head *bh;
} Indirect
```

用数组 chain[4]描述 4 种不同的索引，即直接索引、一级间接索引、二级间接索引、三级间接索引。举例说明这个结构各个域的含义。如果文件内的块号为 8，则不需要间接索引，所以只用 chain[0]一个 Indirect 结构，p 指向直接索引表下标为 8 处，即＆inode->u.ext2_i.i_data[8]；而 key 则持有该表项的内容，即文件块号所对应的设备上的块号（类似于逻辑页面号与物理页面号的对应关系）；bh 为 NULL，因为没有用于间接索引的块。如果文件内的块号为 20，则需要一次间接索引，索引要用 chian[0]和 chain[1]两个表项。第一个表项 chian[0] 中，指针 bh 仍为 NULL，因为这一层没有用于间接索引的数据块；指针 p 指向＆inode->u.ext2_i.i_data[12]，即间接索引的表项；而 key 持有该项的内容，即对应设备的块号。chain[1]中的指针 bh 则指向进行间接索引的块所在的缓冲区，这个缓冲区的内容就是用作间接索引的一个整数数组，而 p 指向这个数组中下标为 8 处，而 key 则持有该项的内容。这样，根据具体索引的深度 depth，数组 chain[]中的最后一个元素，即 chain[depth-1].key，总是持有目标数据块的物理块号。而从 chain [] 中第 1 个元素 chain [0] 到具体索引的最后一个元素 chain [depth-1] ，则提供了具体索引的整个路径，构成了一条索引链，这也是数据名 chain 的由来。

了解了以上基本内容后，我们来看 ext2_get_block（）函数的具体实现代码。

- 首先调用 ext2_block_to_path（）函数，根据文件内的逻辑块号 iblock 计算出这个数据块落在哪个索引区间，要采用几重索引（1 表示直接）。如果返回值为 0，表示出错，因为文件内块号与设备上块号之间至少也得有一次索引。出错的原因可能是文件内块号太大或为负值。
- ext2_get_branch（）函数深化从 ext2_block_to_path（）所取得的结果，而这合在一起基本上完成了从文件内块号到设备上块号的映射。从 ext2_get_branch（）返回的值有两种可能。一是，如果顺利完成了映射则返回值为 NULL。二是，如果在某一索引级发现索引表内的相应表项为 0，则说明这个数据块原来并不存在，现在因为写操作而需要扩充文件的大小。此时，返回指向 Indirect 结构的指针，表示映射在此断裂。此外，如果映射的过程中出错，例如，读数据块失败，则通过 err 返回一个出错代码。
- 如果顺利完成了映射，就把所得结果填入缓冲区结构 bh_result 中，然后把映射过程中读入的缓冲区（用于间接索引）全部释放。
- 可是，如果 ext2_get_branch（）返回一个非 0 指针，那就说明映射在某一索引级上断裂了。根据映射的深度和断裂的位置，这个数据块也许是个用于间接索引的数据块，也许是最终的数据块。不管怎样，此时都应该为相应的数据块分配空间。
- 要分配空间，首先应该确定从物理设备上何处读取目标块。根据分配算法，所分配的数据块应该与上一次已分配的数据块在设备上连续存放。为此目的，在 ext2_inode_info 结构中设置了两个域 i_next_alloc_block 和 i_next_alloc_goal。前者用来记录下一次要分配的文件内块号，而后者则用来记录希望下一次能分配的设备上的块号。在正常情况下，对文件的扩充是顺序的，因此，每次所分配的文件内块号都与前一次的连续，而理想上来说，设

备上的块号也同样连续，二者平行地向前推进。这种理想的“建议块号”就是由 ext2_find_goal（）函数来找的。

- 设备上具体物理块的分配，以及文件内数据块与物理块之间映射的建立，都是调用 ext2_alloc_branch（）函数完成的。调用之前，先要算出还有几级索引需要建立。
- 从 ext2_alloc_branch（）返回以后，我们已经从设备上分配了所需的数据块，包括用于间接索引的中间数据块。但是，原先映射开始断开的最高层上所分配的数据块号只是记录了其 Indirect 结构中的 key 域，却并没有写入相应的索引表中。现在，就要把断开的“树枝”接到整个索引树上，同时，还需要对文件所属 inode 结构中的有关内容做一些调整。这些操作都是由 ext2_splice_branch（）函数完成。

到此为止，万事具备，则转到标号 got_it 处，把映射后的数据块连同设备号置入 bh_result 所指的缓冲区结构中，这就完成了数据块的分配。

第十章　模块机制

如前所述，Linux 是一个整体式的内核（Monolithic Kernel）结构，也就是说，整个内核是一个单独的、非常大的程序，从实现机制来说，我们又把它划分为 5 个子系统（参见第一章内核结构），内核的各个子系统都提供了内部接口（函数和变量），这些函数和变量可供内核所有子系统调用和使用。

另一种是微内核结构，内核的功能块被划分成独立的模块，这些功能块之间有严格的通信机制，要给内核增加一个新成分，配置过程是相当费时的，例如，如果 NCR 810 SCSI 需要一个 SCSI 驱动程序，你不能把它直接加入内核，在用 NCR 810 之前，你就得进行配置并且建立新的内核。

Linux 的整体式结构决定了要给内核增加新的成分也是非常困难，因此 Linux 提供了一种全新的机制——可装入模块（Loadable Modules，以下简称模块），用户可以根据自己的需要，在不需要对内核进行重新编译的条件下，模块能被动态地插入到内核或从内核中移走。

10.1　概述

10.1.1　什么是模块

模块是内核的一部分（通常是设备驱动程序），但是并没有被编译到内核里面去。它们被分别编译并连接成一组目标文件，这些文件能被插入到正在运行的内核，或者从正在运行的内核中移走，进行这些操作可以使用 insmod（插入模块）或 rmmod（移走模块）命令，或者，在必要的时候，内核本身能请求内核守护进程(kerned)装入或卸下模块。这里列出在 Linux 内核源程序中所包括的一些模块。

- 文件系统: minix, xiafs, msdos, umsdos, sysv, isofs, hpfs,smbfs, ext3,nfs, proc 等。
- 大多数 SCSI 驱动程序: (如：aha1542, in2000)。
- 所有的 SCSI 高级驱动程序: disk, tape, cdrom, generic。
- 大多数以太网驱动程序: (非常多，不便于在这儿列出，请查看./Documentation/networking/net-modules.txt)。
- 大多数 CD-ROM 驱动程序：

```
aztcd:      Aztech,Orchid,Okano,Wearnes
cm206:      Philips/LMS CM206
```

gscd: Goldstar GCDR-420
mcd, mcdx: Mitsumi LU005, FX001
optcd: Optics Storage Dolphin 8000AT
sjcd: Sanyo CDR-H94A
sbpcd: Matsushita/Panasonic CR52x, CR56x, CD200,
Longshine LCS-7260, TEAC CD-55A
sonycd535: Sony CDU-531/535, CDU-510/515

- 以及很多其他模块，诸如：
  lp: 行式打印机
  binfmt_elf: elf 装入程序
  binfmt_java: java 装入程序
  isp16: cd-rom 接口
  serial: 串口（tty）

这里要说明的是，Linux 内核中的各种文件系统及设备驱动程序，既可以被编译成可安装模块，也可以被静态地编译进内核的映像中，这取决于内核编译之前的系统配置阶段用户的选择。通常，在系统的配置阶段，系统会给出 3 种选择（Y/M/N），“Y”表示要某种设备或功能,并将相应的代码静态地连接在内核映像中；“M”表示将代码编译成可安装模块，“N”表示不安装这种设备。

10.1.2 为什么要使用模块

按需动态装入模块是非常吸引人的，因为这样可以保证内核达到最小并且使得内核非常灵活，例如，当你可能偶尔使用 VFAT 文件系统，你只要安装(mount) VFAT，VFAT 文件系统就成为一个可装入模块，kerneld 通过自动装入 VFAT 文件系统建立你的 Linux 内核，当你卸下(unmount)VFAT 部分时，系统检测到你不再需要的 FAT 系统模块，该模块自动地从内核中被移走。按需动态装入模块还意味着，你会有更多的内存给用户程序。如前所述，内核所使用的内存是永远不会被换出的，因此，如果你有 100KB 不使用的驱动程序被编译进内核，那就意味着你在浪费 RAM。任何事情都是要付出代价的，内核模块的这种优势是以性能和内存的轻微损失为代价的。

一旦一个 Linux 内核模块被装入，那么它就像任何标准的内核代码一样成为内核的一部分,它和任何内核代码一样具有相同的权限和职责。像所有的内核代码或驱动程序一样,Linux 内核模块也能使内核崩溃。

10.1.3 Linux 内核模块的优缺点

利用内核模块的动态装载性具有如下优点：

- 将内核映像的尺寸保持在最小，并具有最大的灵活性；
- 便于检验新的内核代码，而不需重新编译内核并重新引导。

但是，内核模块的引入也带来了如下问题：

- 对系统性能和内存利用有负面影响；
- 装入的内核模块和其他内核部分一样，具有相同的访问权限，因此，差的内核模块会导致系统崩溃；
- 为了使内核模块访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改这些符号表；
- 有些模块要求利用其他模块的功能，因此，内核要维护模块之间的依赖性；
- 内核必须能够在卸载模块时通知模块，并且要释放分配给模块的内存和中断等资源；
- 内核版本和模块版本的不兼容，也可能导致系统崩溃，因此，严格的版本检查是必需的。

尽管内核模块的引入同时也带来不少问题，但是模块机制确实是扩充内核功能一种行之有效的方法，也是在内核级进行编程的有效途径。

10.2　实现机制

Linux 内核模块机制的实现与内核其他部分的关系并不是很大，相对来说也不是很复杂，但其设计思想是非常值得借鉴的。我们并不准备对其实现函数做一一介绍，在此仅介绍主要的数据结构及实现函数。

10.2.1　数据结构

1．模块符号

如前所述，Linux 内核是一个整体结构，而模块是插入到内核中的插件。尽管内核不是一个可安装模块，但为了方便起见，Linux 把内核也看作一个模块。那么模块与模块之间如何进行交互呢，一种常用的方法就是共享变量和函数。但并不是模块中的每个变量和函数都能被共享，内核只把各个模块中主要的变量和函数放在一个特定的区段，这些变量和函数就统称为符号。到底哪些符号可以被共享？Linux 内核有自己的规定。对于内核模块，在 kernel/ksyms.c 中定义了从中可以“移出”的符号，例如进程管理子系统可以“移出”的符号定义如下：

```
/* process memory management */
EXPORT_SYMBOL(do_mmap_pgoff);
EXPORT_SYMBOL(do_munmap);
EXPORT_SYMBOL(do_brk);
EXPORT_SYMBOL(exit_mm);
EXPORT_SYMBOL(exit_files);
EXPORT_SYMBOL(exit_fs);
EXPORT_SYMBOL(exit_sighand);

    EXPORT_SYMBOL(complete_and_exit);
```

```
EXPORT_SYMBOL (__wake_up) ;
EXPORT_SYMBOL (__wake_up_sync) ;
EXPORT_SYMBOL (wake_up_process) ;
EXPORT_SYMBOL (sleep_on) ;
EXPORT_SYMBOL (sleep_on_timeout) ;
EXPORT_SYMBOL (interruptible_sleep_on) ;
EXPORT_SYMBOL (interruptible_sleep_on_timeout) ;
EXPORT_SYMBOL (schedule) ;
EXPORT_SYMBOL (schedule_timeout) ;
EXPORT_SYMBOL (jiffies) ;
EXPORT_SYMBOL (xtime) ;
EXPORT_SYMBOL (do_gettimeofday) ;
EXPORT_SYMBOL (do_settimeofday) ;
```

你可能对这些变量和函数已经很熟悉。其中宏定义 EXPORT_SYMBOL（）本身的含义是“移出符号”。为什么说是“移出”呢？因为这些符号本来是内核内部的符号，通过这个宏放在一个公开的地方，使得装入到内核中的其他模块可以引用它们。

实际上，仅仅知道这些符号的名字是不够的，还得知道它们在内核映像中的地址才有意义。因此，内核中定义了如下结构来描述模块的符号：

```
struct module_symbol
{
        unsigned long value; ／*符号在内核映像中的地址*/
        const char *name;   /*指向符号名的指针*/
};
```

从后面对 EXPORT_SYMBOL 宏的定义可以看出，连接程序（ld）在连接内核映像时将这个结构存放在一个叫做“__ksymtab”的区段中，而这个区段中所有的符号就组成了模块对外“移出”的符号表，这些符号可供内核及已安装的模块来引用。而其他“对内”的符号则由连接程序自行生成，并仅供内部使用。

与 EXPORT_SYMBOL 相关的定义在 include/linux/module.h 中：

```
    #define __MODULE_STRING_1 (x)     #x
   #define __MODULE_STRING (x)       __MODULE_STRING_1 (x)

#define __EXPORT_SYMBOL (sym, str)                          \
const char __kstrtab_##sym[]                           \
__attribute__ ( (section (".kstrtab") ) )  = str;                 \
const struct module_symbol __ksymtab_##sym             \
__attribute__ ( (section ("__ksymtab") ) )  =                     \
 {  (unsigned long) &sym, __kstrtab_##sym }

#if defined(MODVERSIONS) || !defined(CONFIG_MODVERSIONS)
#define EXPORT_SYMBOL(var)  __EXPORT_SYMBOL(var, __MODULE_STRING(var))
```

下面我们以 EXPORT_SYMBOL(schedule)为例，来看一下这个宏的结果是什么。

首先 EXPORT_SYMBOL(schedule)的定义成了__EXPORT_SYMBOL(schedule,“sched- ule”)。而__EXPORT_SYMBOL（）定义了两个语句，第 1 个语句定义了一个名为__kstrtab_ schedule 的字符串，将字符串的内容初始化为“schedule”，并将其置于内核映像中的.kstrtab 区段，注意这是一个专门存放符号名字符串的区段。第 2 个语句则定义了一个名为__kstrtab_ schedule 的 module_symbol 结构，将其初始化为｛&schedule，__kstrtab_ schedule｝结

构，并将其置于内核映像中的__ksymtab 区段。这样，module_symbol 结构中的域 value 的值就为 schedule 在内核映像中的地址，而指针 name 则指向字符串“schedule”。

2．模块引用(Module Reference)

模块引用是一个不太好理解的概念。 有些装入内核的模块必须依赖其他模块，例如，因为 VFAT 文件系统是 FAT 文件系统或多或少的扩充集，那么，VFAT 文件系统依赖（depend）于 FAT 文件系统，或者说，FAT 模块被 VFAT 模块引用，或换句话说，VFAT 为“父”模块，FAT 为“子”模块。其结构如下：

```
struct module_ref
{
        struct module *dep;          /* “父”模块指针*/
        struct module *ref;          /* “子”模块指针*/
        struct module_ref *next_ref;/*指向下一个子模块的指针*/
};
```

在这里“dep”指的是依赖，也就是引用，而“ref”指的是被引用。因为模块引用的关系可能延续下去，例如 A 引用 B，B 有引用 C，因此，模块的引用形成一个链表。

3．模块

模块的结构为 module，其定义如下：

```
   struct module_persist; /* 待决定 */

struct module
 {
        unsigned long size_of_struct;   /* 模块结构的大小，即 sizeof（module） */
        struct module *next;  /*指向下一个模块 */
        const char *name;   /*模块名，最长为 64 个字符*/
        unsigned long size;  /*以页为单位的模块大小*/

        union
           {
                 atomic_t usecount; /*使用计数，对其增减是原子操作*/
                 long pad;
         } uc;                       /* Needs to keep its size - so says rth */

        unsigned long flags;            /* 模块的标志 */

        unsigned nsyms;         /* 模块中符号的个数 */
        unsigned ndeps;       /* 模块依赖的个数 */
        struct module_symbol *syms; /* 指向模块的符号表,表的大小为 nsyms */

            struct module_ref deps; /*指向模块引用的数组，大小为 ndeps */
        struct module_ref *refs;
        int  (*init) (void) ;       /* 指向模块的 init_module () 函数 */
        void  (*cleanup) (void) ;  /* 指向模块的 cleanup_module () 函数 */
        const struct exception_table_entry *ex_table_start;
        const struct exception_table_entry *ex_table_end;
/*以下域是在以上基本域的基础上的一种扩展，因此是可选的。可以调用 mod_member_
```

present()函数来检查以下域的存在与否。*/

```
        const struct module_persist *persist_start;/*尚未定义*/
        const struct module_persist *persist_end;
        int  (*can_unload) (void) ;
        int runsize                                 /*尚未使用*/
        const char *kallsyms_start;                 /*用于内核调试的所有符号 */
        const char *kallsyms_end;
        const char *archdata_start;                 /* 与体系结构相关的特定数据*/
        const char *archdata_end;
        const char *kernel_data;                    /*保留 */
};
```

其中，moudle 中的状态，即 flags 的取值定义如下：

```
   /* Bits of module.flags.  */

#define MOD_UNINITIALIZED      0 ／*模块还未初始化*/
#define MOD_RUNNING             1 ／*模块正在运行*/
#define MOD_DELETED             2 ／*卸载模块的过程已经启动*/
#define MOD_AUTOCLEAN          4 ／*安装模块时带有此标记，表示允许自动
卸载模块*/
#define MOD_VISITED             8 ／*模块被访问过*/
#define MOD_USED_ONCE           16 ／*模块已经使用过一次*/
#define MOD_JUST_FREED          32 ／*模块刚刚被释放*/
#define MOD_INITIALIZING        64 ／*正在进行模块的初始化*/－／
```

如前所述，虽然内核不是可安装模块，但它也有符号表，实际上这些符号表受到其他模块的频繁引用，将内核看作可安装模块大大简化了模块设计。因此，内核也有一个 module 结构，叫做 kernel_module，与 kernel_module 相关的定义在 kernel／module_c 中：

```
#if defined (CONFIG_MODULES)  || defined (CONFIG_KALLSYMS)

extern struct module_symbol __start___ksymtab[];
extern struct module_symbol __stop___ksymtab[];

extern const struct exception_table_entry __start___ex_table[];
extern const struct exception_table_entry __stop___ex_table[];

extern const char __start___kallsyms[] __attribute__  ( (weak) ) ;
extern const char __stop___kallsyms[] __attribute__  ( (weak) ) ;

struct module kernel_module =
{
        size_of_struct:        sizeof (struct module) ,
        name:                  "",
        uc:                    {ATOMIC_INIT (1) },
        flags:                 MOD_RUNNING,
        syms:                    __start___ksymtab,
        ex_table_start:        __start___ex_table,
        ex_table_end:          __stop___ex_table,
        kallsyms_start:        __start___kallsyms,
        kallsyms_end:          __stop___kallsyms,
};
```

首先要说明的是，内核对可安装模块的的支持是可选的。如果在编译内核代码之前的系

统配置阶段选择了可安装模块，就定义了编译提示 CONFIG_MODULES，使支持可安装模块的代码受到编译。同理，对用于内核调试的符号的支持也是可选的。

凡是在以上初始值未出现的域，其值均为 0 或 NULL。显然，内核没有 init_module()和 cleanup_module()函数，因为内核不是一个真正的可安装模块。同时，内核没有 deps 数组，开始时也没有 refs 链。可是，这个结构的指针 syms 指向__start___ksymtab，这就是内核符号表的起始地址。符号表的大小 nsyms 为 0，但是在系统能初始化时会在 init_module()函数中将其设置成正确的值。

在模块映像中也可以包含对异常的处理。发生于一些特殊地址上的异常，可以通过一种描述结构 exception_table_entry 规定对异常的反映和处理，这些结构在可执行映像连接时都被集中在一个数组中，内核的 exception_table_entry 结构数组就为__start___ex_table[]。当异常发生时，内核的异常响应处理程序就会先搜索这个数组，看看是否对所发生的异常规定了特殊的处理，相关内容请看第四章。

另外，从 kernel_module 开始，所有已安装模块的 module 结构都链在一起成为一条链，内核中的全局变量 module_list 就指向这条链：

```
struct module *module_list = &kernel_module;
```

10.2.2　实现机制的分析

当你新建立了最小内核（如何建立新内核，请看相关的 HOWTO），并且重新启动后，你可以利用实用程序“insmod”和“rmmod”，随意地给内核插入或从内核中移走模块。如果 kerneld 守护进程启动，则由 kerneld 自动完成模块的插拔。有关模块实现的源代码在 /kernel/module.c 中，以下是对源代码中主要函数的分析。

1．启动时内核模块的初始化函数 init_modules()

当内核启动时，要进行很多初始化工作，其中，对模块的初始化是在 main.c 中调用 init_modules()函数完成的。实际上，当内核启动时唯一的模块就为内核本身，因此，初始化要做的唯一工作就是求出内核符号表中符号的个数：

```
*/
  * Called at boot time
  */

void __init init_modules (void)
{
         kernel_module.nsyms = __stop___ksymtab - __start___ksymtab;

        arch_init_modules (&kernel_module) ;
}
```

因为内核代码被编译以后，连接程序进行连接时内核符号的符号结构就“移出”到了 ksymtab 区段，__start___ksymtab 为第 1 个内核符号结构的地址，__stop___ksymtab 为最后一个内核符号结构的地址，因此二者之差为内核符号的个数。其中，arch_init_modules 是与体系结构相关的函数，对 i386 来说，arch_init_modules 在 include/i386/module.h 中

定义为：

```
#define arch_init_modules (x)     do { } while  (0)
```

可见，对 i386 来说，这个函数为空。

2．创建一个新模块

当用 insmod 给内核中插入一个模块时，意味着系统要创建一个新的模块，即为一个新的模块分配空间，函数 sys_create_module()完成此功能，该函数也是系统调用 screate_module（）在内核的实现函数，其代码如下：

```
/*
  * Allocate space for a module.
  */

asmlinkage unsigned long
sys_create_module (const char *name_user, size_t size)
{
         char *name;
         long namelen, error;
         struct module *mod;
         unsigned long flags;

         if  (!capable (CAP_SYS_MODULE) )
                 return -EPERM;
         lock_kernel () ;
         if  ( (namelen = get_mod_name (name_user, &name) )  < 0)  {
                error = namelen;
                 goto err0;
         }
        if  (size < sizeof (struct module) +namelen)  {
                 error = -EINVAL;
                goto err1;
         }
        if  (find_module (name)  != NULL)  {
                error = -EEXIST;
                goto err1;
         }
        if  ( (mod =  (struct module *) module_map (size) )  == NULL)  {
                error = -ENOMEM;
                 goto err1;
         }

         memset (mod, 0, sizeof (*mod) ) ;
         mod->size_of_struct = sizeof (*mod) ;
         mod->name =  (char *)  (mod + 1) ;
         mod->size = size;
        memcpy ( (char*)  (mod+1) , name, namelen+1) ;

         put_mod_name (name) ;

         spin_lock_irqsave (&modlist_lock, flags) ;
```

```
        mod->next = module_list;
        module_list = mod;      /* link it in */
       spin_unlock_irqrestore (&modlist_lock, flags) ;

        error = (long) mod;
        goto err0;
err1:
        put_mod_name (name) ;
err0:
        unlock_kernel () ;
        return error;
}
```

下面对该函数中的主要语句给予解释。

- capable(CAP_SYS_MODULE)检查当前进程是否有创建模块的特权。
- 参数 size 表示模块的大小，它等于 module 结构的大小加上模块名的大小，再加上模块映像的大小，显然，size 不能小于后两项之和。
- get_mod_name（）函数获得模块名的长度。
- find_module（）函数检查是否存在同名的模块，因为模块名是模块的唯一标识。
- 调用 module_map（）分配空间，对 i386 来说，就是调用 vmalloc()函数从内核空间的非连续区分配空间。
- memset（）将分配给 module 结构的空间全部填充为 0，也就是说，把通过 module_map（）所分配空间的开头部分给了 module 结构；然后（module+1）表示从 mod 所指的地址加上一个 module 结构的大小，在此处放上模块的名字；最后，剩余的空间给模块映像。
- 新建 moudle 结构只填充了三个值，其余值有待于从用户空间传递过来。
- put_mod_name()释放局部变量 name 所占的空间。
- 将新创建的模块结构链入 module_list 链表的首部。

3．初始化一个模块

从上面可以看出，sys_create_module（）函数仅仅在内核为模块开辟了一块空间，但是模块的代码根本没有拷贝过来。实际上，模块的真正安装工作及其他的一些初始化工作由 sys_init_module()函数完成，该函数就是系统调用 init_module()在内核的实现代码，因为其代码比较长，为此对该函数的主要实现过程给予描述。

该函数的原型为：

```
asmlinkage long sys_init_module (const char *name_user, struct module *mod_user)
```

其中参数 name_user 为用户空间的模块名，mod_user 为指向用户空间欲安装模块的 module 结构。

该函数的主要操作描述如下。

- sys_create_module（）在内核空间创建了目标模块的 module 结构，但是这个结构还基本为空，其内容只能来自用户空间。因此，初始化函数就要把用户空间的 module 结构拷贝到内核中对应的 module 结构中。但是，由于内核版本在不断变化，因此用户空间 module 结构可能与内核中的 module 结构不完全一样，例如用户空间的 module 结构为 2.2 版，而内核空间的为 2.4 版，则二者的 module 结构就有所不同。为了防止二者的 module 结构在大小上

的不一致而造成麻烦，因此，首先要把用户空间的 module 结构中的 size_of_struct 域复制过来加以检查。从前面介绍的 module 结构可以看出，2.4 版中从 persist_start 开始的域是内核对 module 结构的扩充，用户空间的 module 结构中没有这些域，因此二者 module 结构大小的检查不包括扩充域。

- 通过了对结构大小的检查以后，先把内核中的 module 结构保存在堆栈中作为后备，然后就从用户空间拷贝其 module 结构。复制时是以内核中 module 结构的大小为准的，以免破坏内核中的内存空间。
- 复制过来以后，还要检查 module 结构中各个域的合理性。
- 最后，还要对模块名进行进一步的检查。虽然已经根据参数 name_user 从用户空间拷贝过来了模块名，但是这个模块名是否与用户空间 module 结构中所指示的模块名一致呢？显然，也可能存在不一致的可能，因此还要根据 module 结构的内容把模块映像中的模块名也复制过来，再与原来使用的模块名进行比较。
- 经过以上检查以后，可以从用户空间把模块的映像复制过来了。
- 但是把模块映像复制过来并不是万事大吉，模块之间的依赖关系还得进行修正，因为正在安装的模块可能要引用其他模块中的符号。虽然在用户空间已经完成了对这些符号的连接，但现在必须验证所依赖的模块在内核中还未被卸载。如果所依赖的模块已经不在内核中了，则对目标模块的安装就失败了。在这种情况下，应用程序（例如 insmod）有责任通过系统调用 delete_module()将已经创建的 module 结构从 moudle_list 中删除。
- 至此，模块的安装已经基本完成，但还有一件事要做，那就是启动待执行模块的 init_moudle（）函数，每个模块块必须有一个这样的函数，module 结构中的函数指针 init 就指向这个函数，内核可以通过这个函数访问模块中的变量和函数，或者说，init_moudle（）是模块的入口，就好像每个可执行程序的入口都是 main()一样。

4．卸载模块的函数 sys_delete_module()

卸载模块的系统调用为 delete_module()，其内核的实现函数为 sys_delete_module()，该函数的原型为：

```
asmlinkage long  sys_delete_module (const char *name_user)
```

与前面几个系统调用一样，只有特权用户才允许卸载模块。卸载模块的方式有两种，这取决于参数 name_user，name_user 是用户空间中的模块名。如果 name_user 非空，表示卸载一个指定的模块；如果为空，则卸载所有可以卸载的模块。

（1）卸载指定的模块

一个模块能否卸载，首先要看内核中是否还有其他模块依赖该模块，也就是该模块中的符号是否被引用，更具体地说，就是检查该模块的 refs 指针是否为空。此外，还要判断该模块是否在使用中，即__MOD_IN_USE（）宏的值是否为 0。只有未被依赖且未被使用的模块才可以卸载。

卸载模块时主要要调用目标模块的 cleanup_module()函数，该函数撤销模块在内核中的注册，使系统不再能引用该模块。

一个模块的拆除有可能使它所依赖的模块获得自由，也就是说，它所依赖的模块其 refs 队列变为空，一个 refs 队列为空的模块就是一个自由模块，它不再被任何模块所依赖。

(2) 卸载所有可以卸载的模块

如果参数 name_user 为空，则卸载同时满足以下条件的所有模块。

- 不再被任何模块所依赖。
- 允许自动卸载，即安装时带有 MOD_AUTOCLEAN 标志位。
- 已经安装但尚未被卸载，即处于运行状态。
- 尚未被开始卸载。
- 安装以后被引用过。
- 已不再使用。

以上介绍了 init_module（）、create_module（）、delete_module()三个系统调用在内核的实现机制，还有一个查询模块名的系统调用 query_module()。这几个系统调用是在实现 insmod 及 rmmod 实用程序的过程中被调用的，用户开发的程序一般不应该、也不必要直接调用这些系统调用。

5．装入内核模块 request_module()函数

在用户通过 insmod 安装模块的过程中，内核是被动地接受用户发出的安装请求。但是，在很多情况下，内核需要主动地启动某个模块的安装。例如，当内核从网络中接收到一个特殊的 packet 或报文时，而支持相应规程的模块尚未安装；又如，当内核检测到某种硬件时，而支持这种硬件的模块尚未安装等等，类似情况还有很。在这种情况下，内核就调用 request_module()主动地启动模块的安装。

request_module()函数在 kernel/kmod.c 中：

```
  /**
   * request_module - try to load a kernel module
      * @module_name: Name of module
*
      * Load a module using the user mode module loader. The function returns
      * zero on success or a negative errno code on failure. Note that a
      * successful module load does not mean the module did not then unload
      * and exit on an error of its own. Callers must check that the service
      * they requested is now available not blindly invoke it.
      *
      * If module auto-loading support is disabled then this function
      * becomes a no-operation.
      */
int request_module (const char * module_name)
{
           pid_t pid;
         int waitpid_result;
         sigset_t tmpsig;
        int i;
        static atomic_t kmod_concurrent = ATOMIC_INIT (0) ;
 #define MAX_KMOD_CONCURRENT 50  /* Completely arbitrary value - KAO */
         static int kmod_loop_msg;

         /* Don't allow request_module () before the root fs is mounted! */
         if  ( ! current->fs->root )  {
```

```
                printk (KERN_ERR "request_module[%s]: Root fs not mounted\n", module_name) ;
            return -EPERM;
        }

        /* If modprobe needs a service that is in a module, we get a recursive
         * loop.  Limit the number of running kmod threads to max_threads/2 or
         * MAX_KMOD_CONCURRENT, whichever is the smaller.  A cleaner method
         * would be to run the parents of this process, counting how many times
         * kmod was invoked.  That would mean accessing the internals of the
         * process tables to get the command line, proc_pid_cmdline is static
         * and it is not worth changing the proc code just to handle this case.
         * KAO.
         */
        i = max_threads/2;
        if  (i > MAX_KMOD_CONCURRENT)
                i = MAX_KMOD_CONCURRENT;
        atomic_inc (&kmod_concurrent) ;
        if  (atomic_read (&kmod_concurrent)  > i)  {
                if  (kmod_loop_msg++ < 5)
                        printk (KERN_ERR
                    "kmod: runaway modprobe loop assumed and stopped\n") ;
                atomic_dec (&kmod_concurrent) ;
                return -ENOMEM;
        }

        pid = kernel_thread (exec_modprobe,  (void*)  module_name, 0) ;
        if  (pid < 0)  {
                printk (KERN_ERR "request_module[%s]: fork failed, errno %d\n", module_name,
-pid) ;
                atomic_dec (&kmod_concurrent) ;
                return pid;
        }

        /* Block everything but SIGKILL/SIGSTOP */
        spin_lock_irq (&current->sigmask_lock) ;
        tmpsig = current->blocked;
        siginitsetinv (&current->blocked, sigmask (SIGKILL)  | sigmask (SIGS- TOP) ) ;
        recalc_sigpending (current) ;
        spin_unlock_irq (&current->sigmask_lock) ;

        waitpid_result = waitpid (pid, NULL, __WCLONE) ;
        atomic_dec (&kmod_concurrent) ;

        /* Allow signals again.. */
        spin_lock_irq (&current->sigmask_lock) ;
        current->blocked = tmpsig;
        recalc_sigpending (current) ;
        spin_unlock_irq (&current->sigmask_lock) ;

        if  (waitpid_result != pid)  {
    printk (KERN_ERR "request_module[%s]: waitpid (%d,...)  failed, errno %d\n",
                        module_name, pid, -waitpid_result) ;
```

```
        }
        return 0;
    }
```

对该函数的解释如下。

- 因为 request_module（）是在当前进程的上下文中执行的，因此首先检查当前进程所在的根文件系统是否已经安装。
- 对 request_module（）的调用有可能嵌套，因为在安装过程中可能会发现必须先安装另一个模块。例如，MS-DOS 模块需要另一个名为 fat 的模块，fat 模块包含基于文件分配表（FAT）的所有文件系统所通用的一些代码。因此，如果 fat 模块还不在系统中，那么在系统安装 MS-DOS 模块时，fat 模块也必须被动态链接到正在运行的内核中。因此，就要对嵌套深度加以限制，程序中设置了一个静态变量 kmod_concurrent，作为嵌套深度的计数器，并且还规定了嵌套深度的上限为 MAX_KMOD_CONCURRENT。不过，对嵌套深度的控制还要考虑到系统中对进程数量的限制，即 max_therads，因为在安装的过程中要创建临时的进程。
- 通过了这些检查以后，就调用 kernel_thread()创建一个内核线程 exec_modprobe（）。exec_modprobe（）接受要安装的模块名作为参数，调用 execve()系统调用执行外部程序/sbin/modprobe，然后， modprobe 程序真正地安装要安装的模块以及所依赖的任何模块。
- 创建内核线程成功以后，先把当前进程信号中除 SIGKILL 和 SIGSTOP 以外的所有信号都屏蔽掉，免得当前进程在等待模块安装的过程中受到干扰，然后就通过 waitpid（）使当前进程睡眠等待，直到 exec_modprobe（）内核线程完成模块安装后退出。当前进程被唤醒而从 waitpid（）返回时，又要恢复当前进程原有信号的设置。根据 waitpid（）的返回值可以判断 exec_modprobe（）操作的成功与否。如果失败，就通过 prink()在系统的运行日志“／var/log/message ”中记录一条出错信息。

10.3　模块的装入和卸载

10.3.1　实现机制

有两种装入模块的方法，第 1 种是用 insmod 命令人工把模块插入到内核，第 2 种是一种更灵活的方法，当需要时装入模块，这就是所谓的请求装入。

当内核发现需要一个模块时，例如，用户安装一个不在内核的文件系统时，内核将请求内核守护进程（kerneld）装入一个合适的模块。

内核守护进程（kerneld）是一个标准的用户进程，但它具有超级用户权限。kerneld 通常是在系统启动时就开始执行，它打开 IPC（Inter-Process Communication）到内核的通道，内核通过给 kerneld 发送消息请求执行各种任务。

kerneld 的主要功能是装入和卸载内核模块，但它也具有承担其他任务的能力，例如，当必要时，通过串行链路启动 PPP 链路，不需要时，则关闭它。kerneld 并不执行这些任务，

它通过运行诸如 insmod 这样的程序来做这些工作，kerneld 仅仅是内核的一个代理。

insmod 实用程序必须找到请求装入的内核模块，请求装入的内核模块通常保存在/lib/modules/kernel-version/目录下。内核模块被连接成目标文件，与系统中其他程序不同的是，这种目标文件是可重定位的（它们是 a.out 或 ELF 格式的目标文件）。insmods 实用程序位于／sbin 目录下，该程序执行以下操作。

(1) 从命令行中读取要装入的模块名。

(2) 确定模块代码所在的文件在系统目录树中的位置，即/lib/modules／kernel-version/目录。

(3) 计算存放模块代码、模块名和 module 结构所需要的内存区大小。调用 create_module( )系统调用，向它传递新模块的模块名和大小。

(4) 用 QM_MODULES 子命令反复调用 query_module( )系统调用来获得所有已安装模块的模块名。

(5) 用 QM_SYMBOL 子命令反复调用 query_module( )系统调用来获得内核符号表和所有已经安装到内核的模块的符号表。

(6) 使用内核符号表、模块符号表以及 create_module( )系统调用所返回的地址重新定位该模块文件中所包含的文件的代码。这就意味着用相应的逻辑地址偏移量来替换所有出现的外部符号和全局符号。

(7) 在用户态地址空间中分配一个内存区，并把 module 结构、模块名以及为正在运行的内核所重定位的模块代码的一个拷贝装载到这个内存区中。如果该模块定义了 init_module( )函数，那么 module 结构的 init 域就被设置成该模块的 init_module( )函数重新分配的地址。同理，如果模块定义了 cleanup_module( )函数，那么 cleanup 域就被设置成模块的 cleanup_module( )函数所重新分配的地址。

(8) 调用 init_module( )系统调用，向它传递上一步中所创建的用户态的内存区地址。

(9) 释放用户态内存区并结束。

为了取消模块的安装，用户需要调用/sbin/rmmod 实用程序，它执行以下操作：

(1) 从命令行中读取要卸载的模块的模块名。

(2) 使用 QM_MODULES 子命令调用 query_module( )系统调用来取得已经安装的模块的链表。

(3) 使用 QM_REFS 子命令多次调用 query_module( )系统调用来检索已安装的模块间的依赖关系。如果一个要卸载的模块上面还安装有某一模块，就结束。

(4) 调用 delete_module( )系统调用，向其传递模块名。

10.3.2 如何插入和卸载模块

如前所述，插入和卸载模块的实用程序为 insmod 和 rmmod，在此，我们将介绍在使用这些命令的过程中会遇到的问题，而并不详细介绍其用法，其更详细的使用请用 man 命令进行查看。

只有超级用户才能插入一个模块，其简单的命令如下：

```
insmod serial.o
```

其中，serial.o 为串口的驱动程序。

但是，这条命令执行以后可能会出现错误信息，诸如模块与内核版本不匹配、不认识的符号等。

例如，如果想插入 msdos.o，就可能出现如下信息：

```
msdos.o: unresolved symbol fat_date_UNIX2dos
msdos.o: unresolved symbol fat_add_cluster1
msdos.o: unresolved symbol fat_put_super
...
```

这是因为 msdos.o 引用的这些符号不是由内核“移出”的。为了证实这点，你可以查看/proc/ksyms，从中就可以发现内核移出的所有符号，但找不到“fat_date_unix2dos”符号。那么，怎样才能让这个符号出现在符号列表中呢？从这个符号可以看出，msdos.o 所依赖的模块为 fat.o，于是重新使用 insmod 命令把 fat.o 插入到内核，然后再查看/proc/modules，就会发现有两个模块被装入，并且一个模块依赖于另一个模块：

```
msdos                5632    0  (unused)
fat                  30400   0 [msdos]
```

也许你要问，怎么才能知道所依赖的模块呢？除了从符号名判断外，更有效的方法是使用 depmod 和 modprobe 命令来代替 insmod 命令。

当错误信息为“kernel/module version mismatch”时，说明内核和模块的版本不匹配，这部分内容我们将在后面给予讨论。

通常情况下，当你插入模块时，还需要把参数传递给模块。例如，一个设备驱动程序想知道它所驱动的设备的 I/O 地址和 IRQ，或者一个网络驱动程序想知道你要它进行多少次的诊断跟踪。这里给出一个例子：

```
insmod ne.o  io=0x400  irq=10
```

这里装入的是 NE2000 类的以太网适配器驱动程序，并告诉它以太网适配器的 I/O 地址为 0x400，其所产生的中断为 IRQ 10。

对于可装入模块，并没有标准的参数形式，也几乎没有什么约定。每个模块的编写者可以决定 insmod 可以用什么样的参数。对于 Linux 内核现已支持的模块，Linux HOWTO 文档给出了每种驱动程序的参数信息。

另外，一个常见的错误是试图插入一个不是可装入模块的目标文件。例如，在内核配置阶段，你把 USB 核心模块静态地连接进基本内核中，因此，USB 核心模块就不是可装入模块。该模块的文件名是 usbcore.o，这看起来与可装入模块的文件名 usbcore.o 完全一样，但是你不能用 insmod 命令插入那个文件，否则，出现以下错误信息：

```
usbcore.o: couldn't find the kernel version this module was compiled for
```

这条消息告诉你，insmod 把 usbcore.o 当作一个合法的可装入模块来看待，并查找这个模块曾经被编译的内核版本，但没有找到。但我们知道，真正的原因是这个文件根本就不是一个可安装模块。

从内核卸载一个模块的命令为 rmmod，例如卸载 ne 模块的命令为：

```
rmmod  ne
```

10.4 内核版本

模块的编写与内核版本密切相关，在此，我们既要讨论内核版本与模块版本的兼容性问题，也要讨论内核从 2.0 到 2.2 及从 2.2 到 2.4 的内核 API 变化对编写模块的影响。

10.4.1 内核版本与模块版本的兼容性

可装入模块的编写者必须意识到，可装入模块既独立于内核又依赖于内核，所谓“独立”是因为它可以独立编译，所谓依赖是指它要调用内核或其他已装入模块的函数或变量，因此，内核版本的变化直接影响着曾经编写的模块是否能被新的内核认可。

例如，mydriver.o 是基于 Linux 2.2.1 内核编写和编译的，但是有人想把它装入到 Linux 2.2.2 的内核中，如果 mydriver.o 所调用的内核函数在 2.2.2 中有所变化，那么内核怎么知道内核版本与模块所调用函数的版本不一致呢？

为了解决这个问题，可装入模块的开发者就决定给模块也编以内核的版本号。在上面的例子中，mydriver.o 目标文件的.modinfo 特殊区段就含有“2.2.1”，因为 mydriver.o 的编译使用了来自 Linux 2.2.1 的头文件，因此，当把该驱动程序装入到 2.2.2 内核时，insmod 就会发现不匹配而失败，从而告诉你内核版本不匹配。

但是，Linux 2.2.1 和 2.2.2 之间的一点不兼容是否真的影响 mydriver.o 的执行呢？mydriver.o 仅仅调用了内核中几个函数、访问了几个数据结构，可以肯定地说，这些函数和数据结构并不一定随每个版本的稍微变化而变化。这种版本上的严格限制在一定程度上带来了不便，因为每当有版本变动时，就要重新编译（或从网上下载）许多可安装模块，而这种变动也许并不影响模块的运行，因此应当有其他的办法来解决这个问题。

办法之一，insmod 有一个-f 选项来强迫 insmod 忽略内核版本的不匹配，并把模块插入到任何版本中，但是，你仍然会得到一个版本不匹配的警告信息。

办法之二，就是将版本信息编码进符号名中，例如将版本号作为符号名的后缀。这样如果符号名相同而版本号不一致，insmod 就会认为是不同的符号而不予连接。但是，内核把是否将版本信息编码进符号名中是作为一个可选项 CONFIG_MODVERSIONS 来提供的。如果需要版本信息，就可以在编译内核代码前的系统配置阶段选择这个可选项；如果不需要，就可以在编译模块的源代码时加上-D CONFIG_MODVERSIONS 取消这个选项。

当以符号编码来编译内核或模块时，我们前面介绍的 EXPORT_SYMBOL（）宏定义的形式就有所不同，例如模块最常调用的内核函数 register_chrdev（），其函数名的宏定义的在 C 中为：

```
#define register_chrdev register_chrdev_Rc8dc8350
```

把符号 register_chrdev 定义为 register_chrdev 加上一个后缀，这个后缀就是 register_chrdev（）函数实际源代码的校验和，只要函数的源代码改动一个字符，这个校验和也会发生变化。因此，尽管你在源代码中读到的函数名为 register_chrdev，但 C 的预处理程序知道真正调用的是 register_chrdev_Rc8dc8350。

10.4.2　从版本 2.0 到 2.2 内核 API 的变化

与 Linux 2.0 相比，2.2 在性能、资源的利用、健壮性以及可扩展性等方面已经有了很大的改善，这些改善势必导致内核 API 的变化。所谓内核 API 就是指为进行内核扩展而提供的编程接口，内核的编程指编写驱动程序、文件系统及其他的内核代码。有关驱动程序的内容请参见下一章。

1．用户空间与内核空间之间数据的拷贝

早在 Linux 2.1.x 版本时，Liuns 就提出了一种有效的办法来改善内核空间与用户空间之间数据的拷贝。我们知道，内核空间与用户空间之间数据的拷贝要通过一个缓冲区，在以前的内核中，对这个缓冲区有效性的检查是通过 verify_area()函数的，如果这个缓冲区有效，则调用 memcpy_tofs()把数据从内核空间拷贝到用户空间。但是，verify_area()函数是低效的，因为它必须检查每一个页面，看其是否是一个有效的映射。

在 2.1.x（以及后来的版本）中，取消了对用户空间缓冲区每个页面的检查，取而代之的是用异常来处理非法的缓冲区。这就避免了在 SMP 上的竞争条件及有效性检查。verify_area()函数现在仅仅用来检查缓冲区的范围是否合法，这是一个快速的操作。

因此，如果你要把数据拷贝到用户空间，就使用 copy_to_user()函数，其用法如下：

```
if （ copy_to_user （ubuff, kbuff, length） ） return -EFAULT;
```

这里，ubuff 是用户空间的缓冲区，kbuff 是内核空间的缓冲区，而 length 是要拷贝的字节数。如果 copy_to_user()函数返回一个非 0 值，就意味着某些数据没有被拷贝（由于无效的缓冲区）。在这种情况下，返回-EFAULT 以表示缓冲区是无效的。类似地，从用户空间拷贝到内核空间的用法如下：

```
if （ copy_from_user （kbuff, ubuff, length） ） return -EFAULT;
```

注意，这两个函数都自动调用 verify_area()函数，你没必要自己调用它。

2．文件操作的方法

在内核 2.1.42 版本以后，增加了一个目录高速缓存（dcache）层，这个层加速了目录搜索操作（大约能提高 4 倍），但同时也需要改变文件操作接口。对驱动程序的编写者，这个变化相对比较简单：原来传递给 file_operations 某些方法的参数为 struct inode *，现在改为 struct dentry *。如果你的驱动程序要引用 inode，下面代码就足够了：

```
struct inode *inode = dentry->d_inode;
```

假定 dentry 是目录项的变量名。实际上，有些驱动程序就不涉及 inode，因此可忽略这一步。然而，你必须改变的是，重新声明 file_operations 中的函数。注意，某些方法还是把 inode 而不是 dentry 作为参数来传递。

有些方法甚至没有提供 dentry，仅仅提供了 struct file *，在这种情况下，你可以用下面的代码提取出 dentry：

```
struct dentry *dentry = file->f_dentry;
```

假定 file 是指向 file 指针的变量名。

下面是内核 2.2.x 文件操作的方法：

```
loff_t llseek （struct file *, loff_t, int）;
```

```
ssize_t read  (struct file *, char *, size_t, loff_t *) ;
ssize_t write  (struct file *, const char *, size_t, loff_t *) ;
int readdir  (struct file *, void *, filldir_t) ;
unsigned int poll  (struct file *, struct poll_table_struct *) ;
int ioctl  (struct inode *, struct file *, unsigned int, unsigned long) ;
int mmap  (struct file *, struct vm_area_struct *) ;
int open  (struct inode *, struct file *) ;
int flush  (struct file *) ;
int release  (struct inode *, struct file *) ;
int fsync  (struct file *, struct dentry *) ;
int fasync  (int, struct file *, int) ;
int check_media_change  (kdev_t dev) ;
int revalidate  (kdev_t dev) ;
int lock  (struct file *, int, struct file_lock *) ;
```

在你声明自己的 file_operations 结构时，应当确保把自己的方法放置在与上面一致的位置。不过，还有另外一种我们提到过的方法，其形式如下：

```
static struct file_operations mydev_fops = {
     open:     mydev_open,
     release: mydev_close,
     read:     mydev_read,
     write:    mydev_write,
};
```

gcc 编译程序能够把这些方法放在正确的位置，并把未定义的方法置为 NULL。

另外还值得注意的是，Linux 2.2 中引入了 pread() 和 pwrite()系统调用，这就允许进程可以从一个文件的指定位置进行读和写，这与另一个 lseek()系统调用类似但不完全相同。其不同之处是，pread() 和 pwrite()系统调用能对一个文件进行并发访问。为了对这些新的系统调用进行支持，在 read()和 write()方法中增加了第 4 个（或最后一个）参数，这个参数是指向 offset 的一个指针。作为驱动程序的编写者，你不必关心文件的位置，因此可以忽略这个参数。不过，为了正确性起见，你最好在你的驱动程序中避免使用新的系统调用，就像你不必支持 llseek()方法一样，你也可以在 read()和 write()方法的最顶行增加如下行来避免新旧系统调用的差异：

```
if  (ppos != &file->f_pos)  return  – ESPIPE
```

假定 ppos 是指向 offset 指针的变量名，file 是指向 struct file 结构的变量名。对于 read() 和 write()系统调用，传递给参数 offset 的为 file->f_pos 的地址，而对于 pread() 和 pwrite()系统调用，传递给 offset 的为 ppos 的地址，由此可以很容易地区别这种两种情况。

如果你确实关心文件的位置，那就必须使用和更新由 ppos 所指向的值，以跟踪进程正读在何处。

3．信号的处理

新增加的 signal_pending()函数时的信号的处理更加容易和健壮。2.0 版处理方式是:

```
if  (current->signal & ~current->blocked)
```

2.2 版是：

```
if  ( signal_pending  (current)  )
```

4．I/O 空间映射

任何系统都会有输入输出，因此就会涉及对外部设备的访问。在早期的计算机中，外设通常只有几个寄存器，通过这几个寄存器就可以完成对外设的所有操作。而现在的情况则大不一样，外设通常自带几 MB 的存储器，从自 PCI 总线出现以后，更是如此。所以，必须将外设卡上的存储器映射到内存空间，实际上是虚存空间（3GB 以上）。在以前的 Linux 内核版本中，这样的映射是通过 vremap（）建立的，现在改名为 ioremap()，以更确切地反映其真正的意图。

5．I/O 事件的多路技术

select() 和 poll()系统调用可以让一个进程同时处理多个文件描述符，也就是说可以使进程检测同时等待的多个 I/O 设备，当没有设备准备好时，select()阻塞，其中任一设备准备好时,select()就返回。在 Linux 2.0 中,驱动程序通过在 file_operations 结构中提供 select()方法来支持这种技术，而在 Linux 2.2 中，驱动程序必须提供的是 poll()方法，这种方法具有更大的灵活性。

6．丢弃初始化函数和数据

当内核初始化全部完成以后，就可以丢弃以后不再需要的函数和数据，这意味着存放这些函数和数据的内存可以重新得到使用。但这仅仅应用在编译进内核的驱动程序，而不适合于可安装模块。

定义一个以后要丢弃的变量的形式为：

```
static int mydata __initdata = 0;
```

定义一个以后要丢弃的函数的形式为：

```
__initfunc（void myfunc （void））
{
 }
```

__initdata 和 __initfunc 关键字把代码和数据放在一个特殊的“初始化”区段。较理想的做法应当是，尽可能地把更多的代码和数据放在初始化区段，当然，这里的代码和数据指的是初始化以后（当 init 进程启动时）不再使用的。

7．定时的设定

新增加了一些定时设定函数。Linux 2.0 设定定时是这样的：

```
current->timeout = jiffies + timeout;
schedule （）;
```

Linux 2.2 是：

```
timeout = schedule_timeout （timeout）;
```

同理，如果你需要在一个等待队列上睡眠，但需要定时，Linux 2.0 操作是：

```
current->timeout = jiffies + timeout;
interruptible_sleep_on （&wait）;
```

Linux 2.2 是：

```
timeout = interruptible_sleep_on_timeout （&wait, timeout）;
```

注意，这些新函数返回的是剩余时间的多少。在某些情况下，这些函数在定时时间还没

到就返回。

8．向后兼容的宏

你可以把下面的代码包含进自己编写的代码中，这样就不必费神维护是为 Linux 2.2.x 还是为 Linux 2.0.x 所编译的驱动程序。

```
#include <linux/version.h>
#ifndef KERNEL_VERSION
#  define KERNEL_VERSION (a,b,c)   ( ( (a)  << 16)  +  ( (b)  << 8)  +  (c)
#endif
#if  (LINUX_VERSION_CODE < KERNEL_VERSION (2,1,0) )
#  include <linux/mm.h>
static inline unsigned long copy_to_user  (void *to, const void *from,
                      unsigned long n)
{
    if  ( !verify_area  (VERIFY_WRITE, to, n)  )  return n;
    memcpy_tofs  (to, from, n) ;
    return 0;
 }
 static inline unsigned long copy_from_user  (void *to, const void *from,
                       unsigned long n)
{
     if  ( !verify_area  (VERIFY_READ, from, n)  )  return n;
     memcpy_fromfs  (to, from, n) ;
     return 0;
 }
#  define __initdata
#  define __initfunc (func)  func
#else
#  include <asm/uaccess.h>
#endif
#ifndef signal_pending
#  define signal_pending (p)   (  (p) ->signal & ~ (p) ->blocked )
#endif
```

10.4.3　把内核 2.2 移植到内核 2.4

如果你曾对 Linux 2.0 版比较熟悉，现在要在内核 2.4 版下开发驱动程序，那么在了解了 2.0 到 2.2 内核 API 的变化后，还要了解 2.2 到 2.4 的变化。

1．使用设备文件系统（DevFS）

DevFS 设备文件系统是 Linux 2.4 一个全新的功能，它主要为了有效地管理/dev 目录而开发的。我能知道，UNIX/Linux 中所有的目录都是层次结构，唯独/dev 目录是一维结构（没有子目录），这就直接影响着访问的效率及管理的方便与否。另外，/dev 目录下的节点并不是按实际需要创建的，因此，该目录下存在大量实际不用的节点，但一般也不能轻易删除。

理想的／dev 目录应该是层次的、其规模是可伸缩的。DevFS 就是为达到此目的而设计

的。它在底层改写了用户与设备交互的方式和途径。它会给用户在两方面带来影响。首先，几乎所有的设备名称都做了改变，例如："/dev/hda"是用户的硬盘，现在可能被定位于"/dev/ide0/..."。这一修改方案增大了设备可用的名字空间，且容许USB类和类似设备的系统集成。其次，不再需要用户自己创建设备节点。DevFS的 /dev目录最初是空的，里面特定的文件是在系统启动时、或是加载模块后驱动程序装入时建立的。当模块和驱动程序卸载时，文件就消失了。为保持和旧版本的兼容，可以使用一个用户空间守护程序"devfsd"，以使先前的设备名称能继续使用。目前，DevFS 的使用还只是一个实验性选项，由一个编译选项CONFIG_DEVFS_FS加以选择。

（1）注册和注销字符设备驱动程序

如前所述，一个新的文件系统要加入系统，必须进行注册。那么，一个新的驱动程序要加入系统，也必须进行注册。在下一章我们会看到，我们把设备大体分为字符设备和块设备。字符设备的注册和注销调用 register_chrdev()和 unregister_chrdev()函数。注册了设备驱动程序以后，驱动程序应该调用 devfs_register()登记设备的入口点，所谓设备的入口点就是设备所在的路径名；在注销设备驱动程序之前，应该调用 devfs_unregister()取消注册。

devfs_register()和 devfs_unregister() 函数原型为：

```
devfs_handle_t devfs_register (devfs_handle_t dir, const char *name,
    unsigned int flags,
    unsigned int major, unsigned int minor,
    umode_t mode, void *ops, void *info) ;

void devfs_unregister (devfs_handle_t de) ;
```

其中 devfs_handle_t 表示 DevFS 的句柄（一个结构类型），每个参数的含义如下。

dir：我们要创建的文件所在的 DevFS 的句柄。NULL 意味着这是 DevFS 的根，即/dev。

flags：设备文件系统的标志，缺省值为 DEVFS_FL_DEFAULT。

major：主设备号，普通文件不需要这一参数。

minor：次设备号，普通文件也不需要这一参数。

mode：缺省的文件模式（包括属性和许可权）。

ops：指向 file_operations 或 block_device_operations 结构的指针。

info：任意一个指针，这个指针将被写到 file 结构的 private_data 域。

例如，如果我们要注册的设备驱动程序叫做 DEVICE_NAME，其主设备号为 MAJOR_NR，次设备号为 MINOR_NR，缺省的文件操作为 device_fops，则该设备驱动程序的 init_module（）函数和 cleanup_module（）函数如下：

```
int init_module (void)
{
   int ret;

    if ( (ret = register_chrdev (MAJOR_NR, DEVICE_NAME, &device_fops) ) ==0)
        return ret;
}

void cleanup_module (void)
{
   unregister_chrdev (MAJOR_NR, DEVICE_NAME) ;
```

```
}
```

对以上代码进行改写以支持设备文件系统（假定设备入口点的名字为 DEVICE_ENTRY）。

```
#include <linux/devfs_fs_kernel.h>

devfs_handle_t devfs_handle;

int init_module (void)
{
  int ret;

 if  ( (ret = devfs_register_chrdev (MAJOR_NR, DEVICE_NAME, &device_fops) )  == 0)
     return ret;

 devfs_handle = devfs_register (NULL, DEVICE_ENTRY, DEVFS_FL_DEFAULT,
     MAJOR_NR, MINOR_NR, S_IFCHR | S_IRUGO | S_IWUSR,
     &device_fops, NULL) ;
}

void cleanup_module (void)
 {
   devfs_unregister_chrdev (MAJOR_NR, DEVICE_NAME) ;
   devfs_unregister (devfs_handle) ;
 }
```

（2）在 DevFS 名字空间中创建一个目录

devfs_mk_dir()用来创建一个目录，这个函数返回 DevFS 的句柄，这个句柄用作 devfs_register 的参数 dir。

例如，为了在“/dev/mydevice”目录下创建一个设备设备入口点，则进行如下操作：

```
devfs_handle = devfs_mk_dir (NULL, "mydevice", NULL) ;
devfs_register (devfs_handle, DEVICE_ENTRY, DEVFS_FL_DEFAULT,
    MAJOR_NR, MINOR_NR, S_IFCHR | S_IRUGO | S_IWUSR,
    &device_fops, NULL) ;
```

（3）注册一系列设备入口点

如果一个设备有几个从设备号，就说明同一个设备驱动程序控制了几个不同的设备，例如主 IDE 硬盘的主设备号为 3，但其每个分区都有一个从设备号，例如/dev/had2 的从设备号为 2。在 DevFS 下，每个从次设备号也有一个目录，例如/dev/ide0/,/dev/ide1/等，也就是说，每个从设备号都有一个设备入口点，于是就可以调用 devfs_register_series 来创建一系列的设备入口点。设备入口点的名字以 printf()函数中 format 参数的形式来创建。

注册 DEVICE_NR 设备入口点（从设备号从 MINOR_START 开始）的操作如下：

```
devfs_handle = devfs_mk_dir (NULL, "mydevice", NULL) ;

devfs_register_series (devfs_handle, "device%u", max_device, DEVFS_FL_ DEFAULT,
 MAJOR_NR, MINOR_START, S_IFCHR | S_IRUGO | S_IWUSR,
 &device_fops, NULL) ;
```

（4）块设备

注册和注销块设备的函数为：

```
devfs_register_blkdev()
devfs_unregister_blkdev ()
```

3．使用/proc 文件系统

/proc 是一个特殊的文件系统，其安装点一般都固定为/proc。这个文件系统中所有的文件都是特殊文件，其内容不存在于任何设备上。每当创建一个进程时，系统就以其 pid 为文件名在这个目录下建立起一个特殊文件，使得通过这个文件就可以读／写相应进程的用户空间，而当进程退出时则将此文件删除。

/proc 文件系统中的目录项结构 dentry，在磁盘上没有对应结构，而以内存中的 proc_dir_entry 结构来代替，在 include/linux/proc_fs.h 中定义如下：

```
struct proc_dir_entry {
        unsigned short low_ino;
        unsigned short namelen;
        const char *name;
        mode_t mode;
        nlink_t nlink;
        uid_t uid;
        gid_t gid;
        unsigned long size;
        struct inode_operations * proc_iops;
        struct file_operations * proc_fops;
        get_info_t *get_info;
        struct module *owner;
        struct proc_dir_entry *next, *parent, *subdir;
        void *data;
        read_proc_t *read_proc;
        write_proc_t *write_proc;
       atomic_t count;         /* use count */
           int deleted;            /* delete flag */
        kdev_t  rdev;
 };
```

注册和注销/proc 文件系统的机制已经发生了变化。在 Linux 2.2 中，proc_dir_entry 结构是静态定义和初始化的，而在 Linux 2.4 中，这个数据结构被动态地创建。

(1) 传送的数据小于一个页面的大小

当传送的数据小于一个页面大小时，/proc 文件系统的实现可以通过 proc_dir_entry 中的 read_proc 和 write_proc 方法来实现。

假定我们要注册的/proc 文件系统名为“foo”，在 Linux 2.2 中的代码如下。

foo_proc_entry 结构的初始化：

```
struct proc_dir_entry foo_proc_entry = {
   namelen: 3,
   name : "foo",
   mode : S_IRUGO | S_IWUSR,
   read_proc : foo_read_proc,
   write_proc : foo_write_proc,
};
```

proc 文件系统根节点，即目录项 proc_root 的初始化为：

```
struct proc_dir_entry proc_root = {
      low_ino:        PROC_ROOT_INO,
      namelen:        5,
```

```
            name:          "/proc",
            mode:          S_IFDIR | S_IRUGO | S_IXUGO,
            nlink:         2,
            proc_iops:     &proc_root_inode_operations,
            proc_fops:     &proc_root_operations,
            parent:        &proc_root,
    };
```

注册：

```
proc_register (&proc_root, &foo_proc_entry) ;
```

注销：

```
proc_unreigster (&proc_root, foo_proc_entry.low_ino) ;
```

在 Linux 2.4 中。

注册：

```
struct proc_dir_entry *ent;

if ( (ent = create_proc_entry ("foo", S_IRUGO | S_IWUSR, NULL) )  != NULL)  {
     ent->read_proc = foo_read_proc;
     ent->write_proc = foo_write_proc;
  }
```

注销：

```
remove_proc_entry ("foo", NULL) ;
```

(2) 传送数据大于一个页面大小

当传送数据大于一个页面大小时，/proc 文件系统的实现应当通过完整的 file 结构来实现：

在 Linux 2.2 中。

相关数据结构为：

```
struct file_operations foo_file_ops = {
 ......
};

struct inode_operations foo_inode_ops = {
   default_file_ops : &foo_file_ops;
};

struct proc_dir_entry foo_proc_entry = {
  namelen: 3,
  name : "foo",
  mode : S_IRUGO | S_IWUSR,
  ops : &foo_inode_ops,
};
```

注册为：

```
proc_register (&proc_root, &foo_proc_entry) ;
```

注销为：

```
proc_unreigster (&proc_root, foo_proc_entry.low_ino) ;
```

在 Linux 2.4 中。

相关数据结构为：

```
struct file_operations foo_file_ops = {
```

```
 ......
};

struct inode_operations foo_inode_ops = {
 ......
};
```

注册为：

```
struct proc_dir_entry *ent;

if ((ent = create_proc_entry("foo", S_IRUGO | S_IWUSR, NULL)) != NULL) {
     ent->proc_iops = &foo_inode_ops;
     ent->proc_fops = &foo_file_ops;
}
```

注销为：

```
remove_proc_entry("foo", NULL);
```

3．块设备驱动程序

块设备驱动程序的界面有了很大的变化，新引入了 block_device_operations 结构，缓冲区高速缓存的接口也发生了变化。

(1) 设备注册

在 Linux 2.2 中，块设备与字符设备驱动程序的注册基本相同，都是通过 file_operations 结构进行的。在 Linux 2.4 中，引入了新结构 block_device_operations。例如，块设备的名字为 DEVICE_NAME，主设备号为 MAJOR_NR，则在 Linux 2.2 中如下所述。

数据结构为：

```
struct file_operations device_fops = {
   open : device_open,
   release : device_release,
   read : block_read,
   write : block_write,
   ioctl : device_ioctl,
   fsync : block_fsync,
};
```

注册为：

```
register_blkdev(MAJOR_NR, DEVICE_NAME, &device_fops);
```

在 Linux 2.4 中。

数据结构为：

```
#include <linux/blkpg.h>

struct block_device_operations device_fops = {
open : device_open,
release : device_release,
ioctl : device_ioctl,
};
```

注册为：

```
register_blkdev(MAJOR_NR, DEVICE_NAME, &device_fops);
```

(2) 缓冲区高速缓存接口

在块设备驱动程序中，有一个“请求函数”来处理缓冲区高速缓存的请求。在 Linux 2.2 中，请求函数的注册和定义如下。

函数原型为：

```
void device_request (void) ;
```

注册为：

```
blk_dev[MAJOR_NR].request_fn = &device_request;
```

请求函数的定义为：

```
void device_request (void)
{
 while  (1)  {
     INIT_REQUEST;

     ......

     switch  (CURRENT->cmd)  {
     case READ :
          // read
          break;
     case WRITE :
          // write
          break;
     default :
          end_request (0) ;
          continue;
     }

     end_request (1) ;
   }
}
```

在 Linux 2.4 中。

函数原型为：

```
int device_make_request (request_queue_t *q, int rw, struct buffer_head *sbh) ;
```

注册：

```
blk_queue_make_request (BLK_DEFAULT_QUEUE (MAJOR_NR) ,&device_make_request) ;
```

请求函数的定义为：

```
int device_make_request (request_queue_t *q, int rw, struct buffer_head *sbh)
{
   char *bdata;
    int ret = 0;

    ......

    bdata = bh_kmap (sbh) ;

    switch  (rw)  {
    case READ :
      // read
      break;
    case READA :
```

```
        // read ahead
        break;
      case WRITE :
        // write
        break;
      default :
        goto fail;
      }

      ret = 1;

  fail:
      sbh->b_end_io (sbh, ret) ;
      return 0;
  }
```

其中 request_queue_t 类型的定义请参见下一章，bh_kmap（）函数获得内核映射图。

4．PCI 设备驱动程序

Linux 2.4 包含了具有全部特征的资源管理子系统。它提供了“即插即用”功能， PCI 子系统也随之经历了改变。在 Linux 2.2 中，设备驱动程序搜索所驱动的设备，在 Linux 2.4 中，当驱动程序初始化时就注册设备的信息，当找到一个设备时 PCI 子系统就调用设备的初始化程序。

（1）驱动程序注册

假设要驱动的设备其商家 id 和设备 id 分配为 VENDOR_ID 和 DEVICE_ID，在 Linux 2.2 中，设备初始化函数如下：

```
struct pci_dev *pdev = NULL;

while  ( (pdev = pci_find_device (VENDOR_ID, DEVICE_ID, pdev) )  != NULL)  {
     // initialize each device
 }
```

在 Linux 2.4 中。

数据结构为：

```
struct pci_device_id device_pci_tbl[] __initdata = {
    { VENDOR_ID, DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID },
    { 0, 0, 0, 0 },
};

int device_init_one (struct pci_dev *dev, const struct pci_device_id *ent) ;
void device_remove_one (struct pci_dev *pdev) ;

 struct pci_driver device_driver = {
   name :      DEVICE_NAME,
   id_table :  device_pci_tbl,
   probe :          device_init_one,
   remove :         device_remove_one,
   suspend :        device_suspend,
   resume :         device_resume,
};
```

注册为：

```
if (pci_register_driver (&device_driver) <= 0)
    return -ENODEV;
```

注销为：

```
pci_unregister_driver (&device_driver) ;
```

5．文件系统的移植问题文件系统的移植问题，在此不赘述

6．下半部分（bottom half）处理程序、软中断（softirq）及 tasklets

为了处理硬件中断之外的中断，Linux 2.2 提供了下半部分。Linux 2.4 提供了两种新的机制：软中断及 tasklet。软中断在 SMP 上不是串行化执行，而是同一个处理程序可以在多个 CPU 上同时执行。为了提高 SMP 的性能，软中断机制现在主要用于网络子系统。对于 tasklet 来说，多个 tasklet 可以在多个 CPU 上执行，但一个 CPU 一次只能处理一个 tasklet。下半部分（bh）是由内核串行执行的，即使在 SMP 环境下，一个 CPU 也只能处理一个下半部分。因此，下半部分变得过时，一般情况下，使用 tasklet 就足够了。把下半部分移植到 tasklet 的具体内容请参看第三章的 3.5.6 节。

7．链表及等待队列

（1）通用双向链表

Linux 2.2 是以宏和内联函数的形式来定义通用链表的。Linux 2.2 主要在文件系统中使用了这种链表，而 Linux 2.4 使用得更加普遍（例如等待队列）。

在 include/linux/list.h 中定义的通用链表 list_head 如下：

```
struct list_head {
        struct list_head *next, *prev;
    };
```

如果我们定义一个整型数据的链表，则其定义如下：

```
struct foo_list {
    int data;
    struct list_head list;
};
```

然后，链表的头应该定义如下：

```
LIST_HEAD (foo_list_head) ;
```

或者为

```
struct list_head foo_list_head = LIST_HEAD_INIT (data_list_head) ;
```

或者为：

```
    struct list_head foo_list_head;
INIT_LIST_HEAD (&foo_list_head) ;
```

现在，我们可以用 list_add()为链表增加一个节点，用 list_del()删除一个节点：

```
struct foo_list data;
list_add (&data.list, &foo_list_head) ;
list_del (&data.list) ;
```

使用 list_for_each()和 list_entry()来遍历链表：

```
struct list_head *head, *curr;
```

```
    struct foo_list *element;

    head = &foo_list_head;
    curr = head->next;

    list_for_each (curr, head)
        element = list_entry (curr, struct foo_list, list) ;
```

（2）等待队列

Linux 2.2 以单链表实现了任务的等待队列，而 Linux 2.4 用通用双向链表实现了等待队列。

在 Linux 2.2 中。

队列的定义为：

```
struct wait_queue *wq = NULL;
```

睡眠和唤醒为：

```
interruptible_sleep_on (&wq) ;
wake_up_interruptible (&wq) ;
```

在 Linux 2.4 中，等待队列的定义发生了变化，但实现函数还是一样的。

队列定义为：

```
DECLARE_WAIT_QUEUE_HEAD (wq) ;
```

或者为：

```
wait_queue_head_t wq;
init_waitqueue_head (&wq) ;
```

10.5　编写内核模块

要写一个内核模块，你必须懂得 C 语言，并且你曾经写过一些程序当作进程而运行。但是，编写内核模块和编写一般的程序有很大的不同，首先，你要明白，内核模块一旦插入到内核，它将成为内核的一部分，这意味着仅仅无法控制的一个指针可能导致内核的崩溃；其次，你必须了解一些内核函数的使用方法和功能，尤其是和编写模块相关的函数；最后，要注意内核的版本号，如果你的模块是基于 2.0 版本编写的，你可能希望移植到 Linux 2.2；或者是基于 2.2 版本编写的，而希望移植到 2.4，这就需要知道这些版本的变化对编写模块程序的影响。

Linux 的内核是非常庞大的，但作为一个编程者，你应当至少读一些内核的源文件并弄懂它。然后，编写一个简单的程序，试试到底怎样编写一个内核模块程序。

10.5.1　简单内核模块的编写

一个内核模块应当至少有两个函数，第 1 个为 init_moudle，当模块被插入到内核时调用它；第 2 个为 cleanup_module，当模块从内核移走时调用它。init_module 的主要功能是在内核中注册一个处理某些事的处理程序。cleanup_module 函数的功能是取消 init_module 所做的事情。

下面看一个例子“Hello,world！”。

```
/ * hello.c
  * "Hello,world"   */

/*下面是必要的头文件*/

#include <linux/kernel.h>   /* 内核模块共享这个头文件 */
 #include <linux/module.h>   /* 这是一个模块 */

/* 处理 CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/*初始化模块 */
int init_module ()
{
  printk ("Hello, world - this is a simple module\n") ;

  /* 如果返回一个非 0，那就意味着 init_module 失败，不能装载该内核模块*/
  return 0;
}


/* 取消 init_module 所作的工作*/
void cleanup_module ()
{
  printk ("the module exits the kernel\n") ;
}
```

10.5.2 内核模块的 Makefiles 文件

内核模块不是独立的可执行文件，但在运行时其目标文件被连接到内核中，因此，编译内核模块时必须加 -c 标志，另外， 还得加确定的预定义符号。

__KERNEL__ -- 相当于告诉头文件，这个代码必须运行在内核模式下，而不是用户进程的一部分。

MODULE -- 这个标志告诉头文件，要给出适当的内核模块的定义。

LINUX -- ，从技术上讲，这个标志不是必要的。但是，如果你希望写一个比较正规的内核模块，在多个操作系统上能进行编译，这个标志将会使你感到方便。它可以允许你在独立于操作系统的部分进行常规的编译。

还有其他的一些标志是否被包含进去，这取决于编译模块时的选项。如果你不能明确内核怎样被编译，可以在 in/usr/include/linux/config.h 中查到。

__SMP__ -- ，对称多处理机。如果内核被编译成支持对称多处理机（即使它只不过运行在单个 CPU 上），这必须被定义。如果你要用对称多处理机，还有一些其他的事情必须做，在此不进行详细的讨论。

CONFIG_MODVERSIONS -- ，如果 CONFIG_MODVERSIONS 被激活，当编译内核模块时，你必须定义它，并且包含进 usr/include/linux/modversions.h 中，这也可以由代码本身来做。

Makefile 举例

```
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:  hello.c /usr/include/linux/version.h
          $（CC） $（MODCFLAGS） -c hello.c
          echo insmod hello.o to turn it on
          echo rmmod hello to turn it off
          echo
```

现在，你以 root 的身份对这个内核模块进行编译并连接后，形成一个目标文件 hello.o，然后用 insmod 把 hello 插入到内核，也可以用 rmmod 命令把 hello 从内核移走。如果你想知道结果如何，你可以查看/proc/modules 文件，从中会找到一个新加入的模块。

10.5.3　内核模块的多个文件

有时，可以从逻辑上把内核模块分成几个源文件，在这种情况下，需要做以下事情。

（1）除了一个源文件外，在其他所有的源文件中都要增加一行 #define __NO_VERSION__，这是比较重要的，因为 module.h 通常包括了对 kernel_version 的定义，kernel_version 是一个具有内核版本信息的全局变量，并且编译模块时要用到它。如果你需要 version.h，你就必须自己包含它，但如果你定义了__NO_VERSION__，module.h 就不会被包含进去。

（2）像通常那样编译所有的源文件。

（3）把所有的目标文件结合到一个单独文件中。在 x86 下，这样连接：

```
ld -m elf_i386 -r -o <name of module>.o <第 1 个源文件>.o <第 2 个源文件>.o
```

请看下面例子 start.c。

```
 /* start.c
 *
 * "Hello, world"
 * 这个文件包含了启动例程
 */
/*下面是必要的头文件 */

/* 内核模块的标准形式*/
#include <linux/kernel.h>
#include <linux/module.h>

/* 处理 CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 初始化模块 */
int init_module ()
{
```

```
  printk ("Hello, world - this is the kernel speaking\n") ;

  return 0;
}
```

另一个例子 stop.c。

```
/* stop.c */
/* 这个文件仅仅包含 stop 例程。*/

/* 必要的头文件*/

#include <linux/kernel.h>

#define __NO_VERSION__
#include <linux/module.h>
#include <linux/version.h>

#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

void cleanup_module ()
{
  printk ("Short is the life of a kernel module\n") ;
}
```

下面是多个文件的 Makefile。

```
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:  start.o stop.o
          ld -m elf_i386 -r -o hello.o start.o stop.o

start.o:  start.c /usr/include/linux/version.h
          $ (CC)  $ (MODCFLAGS)  -c start.c

stop.o:       stop.c /usr/include/linux/version.h
          $ (CC)  $ (MODCFLAGS)  -c stop.c
```

“hello”是模块名，它占用了一页（4KB）的内存，此时，没有其他内核模块依赖它。要从内核移走这个模块，敲入“rmmod hello”，注意，rmmod 命令需要的是模块名而不是文件名。其他实用程序的使用可参看相关的文档。

关于模块编程更多的内容我们将在后续章节继续讨论。

第十一章　设备驱动程序

操作系统的主要任务之一是控制所有的输入/输出设备。它必须向设备发布命令，捕获中断并进行错误处理，它还要提供一个设备与系统其余部分的简单易用的界面，该界面应该对所有的设备尽可能的一致，从而将系统硬件设备细节从用户视线中隐藏起来，例如虚拟文件系统对各种已安装的文件系统类型提供了统一的视图而屏蔽了具体底层细节，具体细节都是由设备驱动程序来完成的，对于驱动程序，在 Linux 中可以按照模块的形式进行编译和加载。

11.1　概述

在 Linux 中输入/输出设备被分为 3 类：块设备，字符设备和网络设备。这种分类的使用方法，可以将控制不同输入/输出设备的驱动程序和其他操作系统软件成分分离开来。例如文件系统仅仅控制抽象的块设备，而将与设备有关的部分留给低层软件，即驱动程序。字符设备指那些无需缓冲区可以直接读写的设备，如系统的串口设备/dev/cua0 和/dev/cua1。块设备则仅能以块为单位进行读写的设备，如软盘、硬盘、光盘等，典型块的大小为 512 或 1024 字节。从名称使人想到，字符设备在单个字符的基础上接收和发送数据。为了改进传送数据的速度和效率，块设备在整个数据缓冲区填满时才一起传送数据。网络设备可以通过 BSD 套接口访问数据，关于这方面的内容我们将在第十二章中进行讨论。

在 Linux 中，对每一个设备的描述是通过主设备号和从设备号，其中主设备号描述控制这个设备的驱动程序，也就是说驱动程序和主设备号是一一对应的，从设备号是用来区分同一个驱动程序控制的不同设备。例如主 IDE 硬盘的每个分区的从设备号都不相同，/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主、从设备号将包含在系统调用中的设备特殊文件映射到设备的管理程序，以及大量系统表格中，如字符设备表—chrdevs。 块（磁盘）设备和字符设备的设备特殊文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但 Linux 寻找和初始化网络设备时才建立这种文件。

11.1.1　I/O 软件

I/O 软件的总体目标就是将软件组织成一种层次结构，低层软件用来屏蔽具体设备细节，高层软件则为用户提供一个简洁规范的界面。这种层次结构很好地体现了 I/O 设计的一个关键的概念：设备无关性，其含义就是程序员写的软件无需须修改就能读出软盘，硬盘以及

CD-ROM 等不同设备上的文件。

输入/输出系统的层次结构及各层次的功能如图 11.1 所示。

从图可以看出，用户进程的下层是设备无关的软件，在 Linux 中，设备无关软件的功能大部分由文件系统去完成，其基本功能就是执行适用于所有设备的常用的输入/输出功能，向用户软件提供一个一致的接口。其结构如图 11.2 所示。

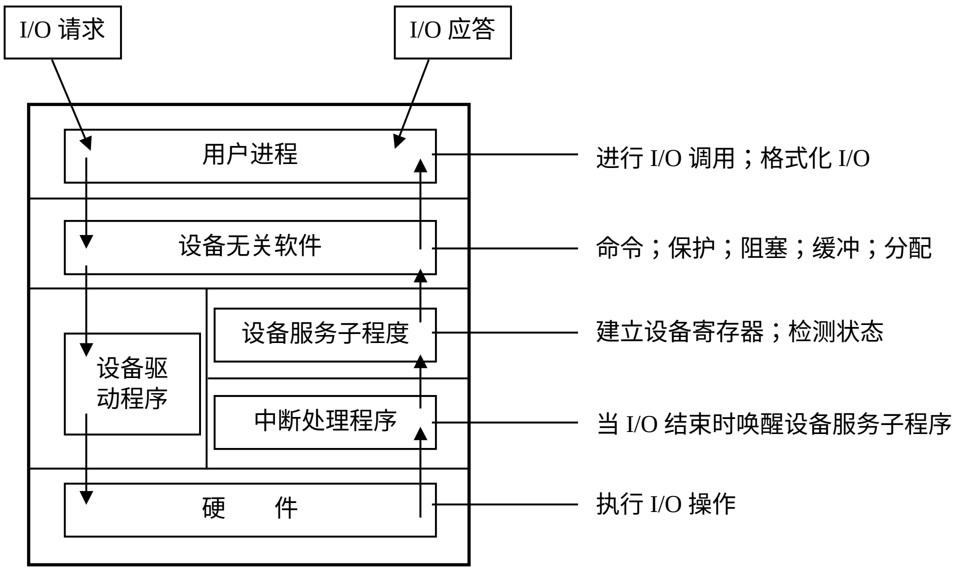


图 11.1　输入/输出系统的层次结构及各层次的功能

对设备程序的统一接口
设备命名
设备保护
提供一个独立于设备的块
缓冲
块设备的存储分配
分配和释放独占设备
错误报告

图 11.2　设备无关软件的功能

设备无关的软件具有以下特点。

- 文件和设备采用统一命名。设备无关软件负责将设备名映射到相应的驱动程序，一个设备名唯一地确定一个索引节点，索引节点中包含了主设备号和从设备号，通过主设备号可以找到相应的设备驱动程序，通过从设备号确定具体的物理设备。

- 对设备提供的保护机制同文件系统一样都采用 rwx 权限。
- 数据块的大小可能对于不同的设备其大小不一样，但操作系统屏蔽这一事实，向高层软件提供了统一的逻辑块的大小。
- 为了解决数据交换速度的匹配问题，采用了缓冲技术，对于缓冲区的管理由文件系统去完成。
- 块设备的存储分配也是由文件系统去处理。
- 对于独占设备的分配和释放属于对临界资源的管理。

11.1.2　设备驱动程序

CPU 并不是系统中唯一的智能设备，每个物理设备都拥有自己的控制器。键盘、鼠标和串行口由一个高级 I/O 芯片统一管理，IDE 控制器控制 IDE 硬盘，而 SCSI 控制器控制 SCSI 硬盘等等。每个硬件控制器都有各自的控制状态寄存器（CSR）并且各不相同。例如 Adaptec 2940 SCSI 控制器的 CSR 与 NCR 810 SCSI 控制器完全不一样。这些寄存器用来启动、停止、初始化设备以及对设备进行诊断。在 Linux 中管理硬件设备控制器的代码并没有放置在每个应用程序中而是由内核统一管理，这些处理和管理硬件控制器的软件就是设备驱动程序。Linux 内核的设备管理是由一组运行在特权级上，驻留在内存以及对底层硬件进行处理的共享库的驱动程序来完成的。

设备管理的一个基本特征是设备处理的抽象性，即所有硬件设备都被看成普通文件，可以通过用操纵普通文件相同的系统调用来打开、关闭、读取和写入设备。系统中每个设备都用一种设备特殊文件来表示，例如系统中第一个 IDE 硬盘被表示成/dev/hda。

那么，系统是如何将设备在用户视野中屏蔽起来的呢？图 11.3 说明了用户进程请求设备进行输入输出的简单流程。

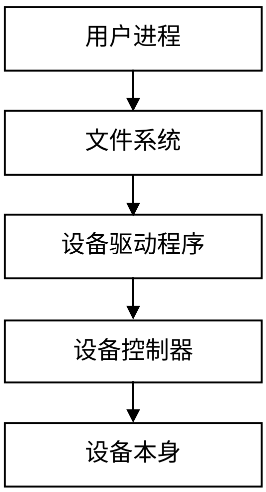


图 11.3　用户进程请求设备服务的流程

首先当用户进程发出输入输出时，系统把请求处理的权限放在文件系统，文件系统通过驱动程序提供的接口将任务下放到驱动程序，驱动程序根据需要对设备控制器进行操作，设

备控制器再去控制设备本身。

这样通过层层隔离，对用户进程基本上屏蔽了设备的各种特性，使用户的操作简便易行，不必去考虑具体设备的运作，就像对待文件操作一样去操作设备，因为实际上在驱动程序向文件系统提供的接口已经屏蔽掉了设备的电器特性。

设备控制器对设备本身的控制是电器工程师所关心的事情，操作系统对输入/输出设备的管理只是通过文件系统和驱动程序来完成的。也就是说在操作系统中，输入/输出系统所关心的只是驱动程序。

Linux 设备驱动程序的主要功能有：

- 对设备进行初始化；
- 使设备投入运行和退出服务；
- 从设备接收数据并将它们送回内核；
- 将数据从内核送到设备；
- 检测和处理设备出现的错误。

在 Linux 中，设备驱动程序是一组相关函数的集合。它包含设备服务子程序和中断处理程序。设备服务子程序包含了所有与设备相关的代码，每个设备服务子程序只处理一种设备或者紧密相关的设备。其功能就是从与设备无关的软件中接受抽象的命令并执行之。当执行一条请求时，具体操作是根据控制器对驱动程序提供的接口(指的是控制器中的各种寄存器)，并利用中断机制去调用中断服务子程序配合设备来完成这个请求。设备驱动程序利用结构 file_operations 与文件系统联系起来，即设备的各种操作的入口函数存在 file_operation 中。对于特定的设备来说有一些操作是不必要的，其入口置为 NULL。

Linux 内核中虽存在许多不同的设备驱动程序但它们具有一些共同的特性，如下所述。

1．驱动程序属于内核代码

设备驱动程序是内核的一部分，它像内核中其他代码一样运行在内核模式，驱动程序如果出错将会使操作系统受到严重破坏，甚至能使系统崩溃并导致文件系统的破坏和数据丢失。

2．为内核提供统一的接口

设备驱动程序必须为 Linux 内核或其他子系统提供一个标准的接口。例如终端驱动程序为 Linux 内核提供了一个文件 I/O 接口。

3．驱动程序的执行属于内核机制并且使用内核服务

设备驱动可以使用标准的内核服务如内存分配、中断发送和等待队列等。

4．动态可加载

多数 Linux 设备驱动程序可以在内核模块发出加载请求时加载，而不再使用时将其卸载。这样内核能有效地利用系统资源。

5. 可配置

Linux 设备驱动程序可以连接到内核中。当内核被编译时，被连入内核的设备驱动程序是可配置的。

11.2 设备驱动基础

11.2.1 I/O 端口

每个连接到 I/O 总线上的设备都有自己的 I/O 地址集，即所谓的 I/O 端口（I/O port）。在 IBM PC 体系结构中，I/O 地址空间一共提供了 65,536 个 8 位的 I/O 端口。可以把两个连续的 8 位端口看成一个 16 位端口，但是这必须是从偶数地址开始。同理，也可以把两个连续的 16 位端口看成一个 32 位端口，但是这必须是从 4 的整数倍地址开始。有 4 条专用的汇编语言指令可以允许 CPU 对 I/O 端口进行读写：它们分别是 in、ins、out 和 outs。在执行其中的一条指令时，CPU 使用地址总线选择所请求的 I/O 端口，使用数据总线在 CPU 寄存器和端口之间传送数据。

I/O 端口还可以被映射到物理地址空间，因此，处理器和 I/O 设备之间的通信就可以直接使用对内存进行操作的汇编语言指令（例如，mov、and、or 等等）。现代的硬件设备更倾向于映射 I/O，因为这样处理的速度较快，并可以和 DMA 结合起来使用。

系统设计者的主要目的是提供对 I/O 编程的统一方法，但又不牺牲性能。为了达到这个目的，每个设备的 I/O 端口都被组织成如图 11.4 所示的一组专用寄存器。CPU 把要发给设备的命令写入控制寄存器（Control Register），并从状态寄存器（Status Register）中读出表示设备内部状态的值。CPU 还可以通过读取输入寄存器（Input Register）的内容从设备取得数据，也可以通过向输出寄存器（Output Register）中写入字节而把数据输出到设备。

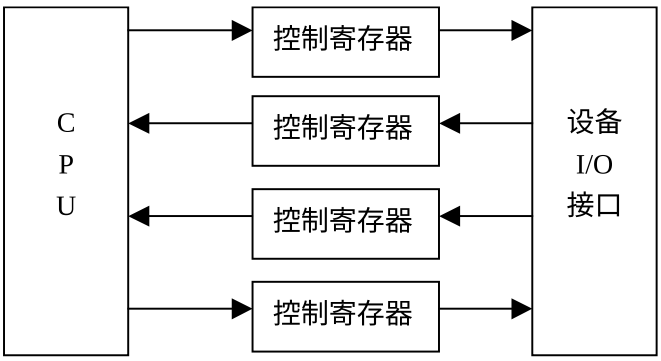


图 11.4 专用 I/O 端口

为了降低成本，通常把同一 I/O 端口用于不同目的。例如，某些位描述设备的状态，而其他位指定发布给设备的命令。同理，也可以把同一 I/O 端口用作输入寄存器或输出寄存器。

那么如何访问 I/O 端口？in、out、ins 和 outs 汇编语言指令都可以访问 I/O 端口。Linux 内核中定义了以下辅助函数来简化这种访问。

1．inb( )、inw( )、inl( )函数

分别从 I/O 端口读取 1、2 或 4 个连续字节。后缀“b”、“w”、“l”分别代表一个字节（8 位）、一个字（16 位）以及一个长整型（32 位）。

2．inb_p( )、inw_p( )、inl_p( )

分别从 I/O 端口读取 1、2 或 4 个连续字节，然后执行一条“哑元（dummy，即空指令）”指令使 CPU 暂停。

3．outb( )、outw( )、outl( )

分别向一个 I/O 端口写入 1、2 或 4 个连续字节。

4．outb_p( )、outw_p( )、outl_p( )

分别向一个 I/O 端口写入 1、2 或 4 个连续字节，然后执行一条“哑元”指令使 CPU 暂停。

5．insb( )、insw( )、insl( )

分别从 I/O 端口读入以 1、2 或 4 个字节为一组的连续字节序列。字节序列的长度由该函数的参数给出。

6．outsb( )、outsw( )、outsl( )

分别向 I/O 端口写入以 1、2 或 4 个字节为一组的连续字节序列。

虽然访问 I/O 端口非常简单，但是检测哪些 I/O 端口已经分配给 I/O 设备可能就不这么简单，特别是对基于 ISA 总线的系统来说更是如此。通常，I/O 设备驱动程序为了侦探硬件设备，需要盲目地向某一 I/O 端口写入数据；但是，如果其他硬件设备已经使用这个端口，那么系统就会崩溃。为了防止这种情况的发生，内核必须使用 iotable 表来记录分配给每个硬件设备的 I/O 端口。任何设备驱动程序都可以使用下面 3 个函数。

request_region( )

把一个给定区间的 I/O 端口分配给一个 I/O 设备。

check_region( )

检查一个给定区间的 I/O 端口是否空闲，或者其中一些是否已经分配给某个 I/O 设备。

release_region( )

释放以前分配给一个 I/O 设备的给定区间的 I/O 端口。

当前分配给 I/O 设备的 I/O 地址可以从/proc/ioports 文件中获得。

11.2.2　I/O 接口及设备控制器

I/O 接口是处于一组 I/O 端口和对应的设备控制器之间的一种硬件电路。它起翻译器的作用，即把 I/O 端口中的值转换成设备所需要的命令和数据。从另一个角度来看，它检测设备状态的变化，并对起状态寄存器作用的 I/O 端口进行相应地更新。还可以通过一条 IRQ 线把这种电路连接到可编程中断控制器上，以使它代表相应的设备发出中断请求。

有两类类型的接口，如下所述。

1．专用 I/O 接口

专门用于一个特定的硬件设备。在一些情况下，设备控制器与这种 I/O 接口处于同一块卡中，连接到专用 I/O 接口上的设备可以是内部设备（位于 PC 机箱内部的设备），也可以是外部设备（位于 PC 机箱外部的设备）。例如键盘接口、图形接口、磁盘接口、总线鼠标接口及网络接口都属于专用 I/O 接口。

2．通用 I/O 接口

用来连接多个不同的硬件设备。连接到通用 I/O 接口上的设备通常都是外部设备。例如并口、串口、通用串行总线（USB）、PCMCIA 接口及 SCSI 接口都属于通用 I/O 接口。

复杂的设备可能需要一个设备控制器来驱动。控制器具有两方面的作用，一是对从 I/O 接口接收到的高级命令进行解释，并通过向设备发送适当的电信号序列强制设备执行特定的操作；二是对从设备接收到的电信号进行转换和解释，并通过 I/O 接口修改状态寄存器的值。

磁盘控制器是一种比较典型的设备控制器，它通过 I/O 接口从微处理器接收诸如“写这个数据块”之类的高级命令，并将其转换成诸如“把磁头定位在正确位置的磁道”上和“把数据写入这个磁道”之类的低级磁盘操作。现在的磁盘控制器相当复杂，因为它们可以把磁盘数据快速保存到内存的缓存区中，还可以根据实际磁盘的几何结构重新安排 CPU 的高级请求，使其优化。

11.2.3　设备文件

设备文件是用来表示 Linux 所支持的大多数设备的，每个设备文件除了设备名，还有 3 个属性：即类型、主设备号、从设备号。

设备文件是通过 mknod 系统调用创建的。其原型为：

```
mknod(const char * filename, int mode, dev_t dev)
```

其参数有设备文件名、操作模式、主设备号及从设备号。最后两个参数合并成一个 16 位的 dev_t 无符号短整数，高 8 位用于主设备号，低 8 位用于从设备号。内核中定义了 3 个宏来处理主、从设备号：MAJOR 和 MINOR 宏可以从 16 位数中提取出主、从设备号，而 MKDEV 宏可以把主、从号合并为一个 16 位数。实际上，dev_t 是专用于应用程序的一个数据类型；在内核中使用 kdev_t 数据类型。在 Linux 2.4 及以前的版本中，这两个类型都会是一个无符号短整型，但是在以后的 Linux 版本中，kdev_t 会成为一个完整的设备文件描述符，也就说，也许会扩成 32 位的长整数。

分配给设备号的正式注册信息及/dev 目录索引节点存放在 documentation/devices.txt 文件中。也可以在 include/linux/major.h 文件中找到所支持的主设备号。

设备文件通常位于/dev 目录下。表 11.1 显示了一些设备文件的属性。注意同一主设备号既可以标识字符设备，也可以标识块设备。

表 11.1　　设备文件的例子

设备名	类型	主设备号	从号	说明
/dev/fd0	块设备	2	0	软盘
/dev/hda	块设备	3	0	第 1 个 IDE 磁盘
/dev/hda2	块设备	3	2	第 1 个 IDE 磁盘上的第 2 个主分区
/dev/hdb	块设备	3	64	第 2 个 IDE 磁盘
/dev/hdb3	块设备	3	67	第 2 个 IDE 磁盘上的第 3 个主分区
/dev/ttyp0	字符设备	3	0	终端
/dev/console	字符设备	5	1	控制台
/dev/lp1	字符设备	6	1	并口打印机
/dev/ttyS0	字符设备	4	64	第 1 个串口
/dev/rtc	字符设备	10	135	实时时钟
/dev/null	字符设备	1	3	空设备（黑洞）

一个设备文件通常与一个硬件设备（如硬盘，/dev/hda）相关连， 或硬件设备的某一物理或逻辑分区（如磁盘分区，/dev/hda2）相关联。但在某些情况下，设备文件不会和任何实际的硬件关联，而是表示一个虚拟的逻辑设备。例如，/dev/null 就是对应于一个“黑洞”的设备文件：所有写入这个文件的数据都被简单地丢弃，因此，该文件看起来总为空。

就内核所关心的内容而言，设备文件名是无关紧要的。如果你建立了一个名为/tmp/disk 的设备文件，类型为“块”，主设备号是 3，从设备号为 0，那么这个设备文件就和表中的/dev/hda 等价。另一方面，对某些应用程序来说，设备文件名可能就很有意义。例如，通信程序可以假设第 1 个串口和/dev/ttyS0 设备文件关联。

1．块设备和字符设备的比较

块设备具有以下特点。

- 可以在一次 I/O 操作中传送固定大小的数据块。
- 可以随机访问设备中所存放的块：传送数据块所需要的时间独立于块在设备中的位置，也独立于当前设备的状态。

块设备典型的例子是硬盘、软盘及 CD-ROM。也可以把 RAM 磁盘当作块设备来对待，这是通过把部分 RAM 配置成快速硬盘而获得的，因此，可以把这部分 RAM 作为应用程序高效存取数据的临时存储器。

字符设备具有以下特点。

- 可以在一次 I/O 操作中传送任意大小的数据。实际上，诸如打印机之类的字符设备可以一次传送一个字节，而诸如磁带之类的设备可以一次传送可变大小的数据块。
- 通常访问连续的字符。

2．网卡

有些 I/O 设备没有对应的设备文件。最明显的一个例子是网卡。实际上，网卡把向外发送的数据放入通往远程计算机系统的一条线上，把从远程系统中接收到的报文装入内核内存。

从 BSD 开始，所有的 UNIX 类系统为计算机中的每个网卡都分配一个不同的符号名。例如，第一个以太网卡名为 eth0。然而，这个名字并没有对应的设备文件，也没有对应的索引节点。

由于没有使用文件系统，所以系统管理员必须建立设备名和网络地址之间的联系。因此，应用程序和网络接口之间的数据通信不是基于标准的有关文件的系统调用的，而是基于 socket( )、bind( )、listen( )、accept()和 connect( )系统调用的，这些系统调用对网络地址进行操作。这组系统调用是在 UNIX BSD 中首先引入的，现在已经成为网络设备的标准编程模型。

11.2.4　VFS 对设备文件的处理

虽然设备文件也在系统的目录树中，但是它们和普通文件以及目录有根本的不同。当进程访问普通文件（即磁盘文件）时，它会通过文件系统访问磁盘分区中的一些数据块。而在进程访问设备文件时，它只要驱动硬件设备就可以了。例如，进程可以访问一个设备文件以从连接到计算机的温度计读取房间的温度。VFS 的责任是为应用程序隐藏设备文件与普通文件之间的差异。

为了做到这点，VFS 改变打开的设备文件的缺省文件操作。因此，可以把对设备文件的任一系统调用转换成对设备相关的函数的调用，而不是对主文件系统相应函数的调用。设备相关的函数对硬件设备进行操作以完成进程所请求的操作。

控制 I/O 设备的一组设备相关的函数称为设备驱动程序。由于每个设备都有一个唯一的 I/O 控制器，因此也就有唯一的命令和唯一的状态信息，所以大部分 I/O 设备类型都有自己的驱动程序。

11.2.5　中断处理

设备一般都比 CPU 慢得多。因此一般情况下，当一个进程通过设备驱动程序向设备发出读写请求后，CPU 并不等待 I/O 操作的完成，而是让正在执行的进程去睡眠，CPU 自己做别的事情，例如唤醒另一个进程执行。当设备完成 I/O 操作需要通知 CPU 时，会向 CPU 发出一个中断请求；然后 CPU 根据中断请求来决定调用相应的设备驱动程序。

当设备执行某个命令时，如“将读取磁头移动到软盘的第 42 扇区上”，设备驱动程序可以从查询方式和中断方式中选择一种来判断设备是否已经完成此命令。

查询方式意味着需要经常读取设备的状态，一直到设备状态表明请求已经完成为止。如

果设备驱动程序被连接进内核，这时使用查询方式将会带来灾难性后果：内核将在此过程中无所事事，直到设备完成目前的请求。有一种方法可以有效地改善这一弊端，就是通过使用系统定时器，使内核周期性调用设备驱动程序中的某个例程来检查设备状态。 使用定时器是查询方式中最好的一种，但更有效的方法是使用中断。

基于中断的设备驱动程序，指的是在硬件设备需要服务时向 CPU 发一个中断信号，引发中断服务子程序执行 。这样就大大地提高了系统资源的利用率，使内核不必一直等到设备执行完任务后才开始有事可干，而是在设备工作期间内核就可以转去处理其他的事务，收到中断请求信号时再回头响应设备。

1. Linux 对中断的管理

Linux 内核为了将来自硬件设备的中断传递到相应的设备驱动程序，在驱动程序初始化的时候就将其对应的中断程序进行了登记，即通过调用函数 request_irq（ ）将其中断信息添加到结构为 irqaction 的数组中，从而使中断号和中断服务程序联系起来。请参见第四章。

request_irq（ ）函数原形如下：

```
int request_irq(unsigned int irq,                /*  中断请求号  */
 void (*handler)(int, void *, struct pt_regs *),/*  指向中断服务子程序 */
      unsigned long irqflags,                     /*  中断类型 */
       const char * devname,                      /*  设备的名字 */
       void *dev_id)；
```

另外，irqaction 的数据结构如下，其图示如图 11.5 所示。

```
struct irqaction {
     void (*handler)(int, void *, struct pt_regs *);
     unsigned long flags;
     unsigned long mask;
     const char *name;
     void *dev_id;
    struct irqaction *next;
};
   static struct irqaction *irq_action[NR_IRQS+1]
```

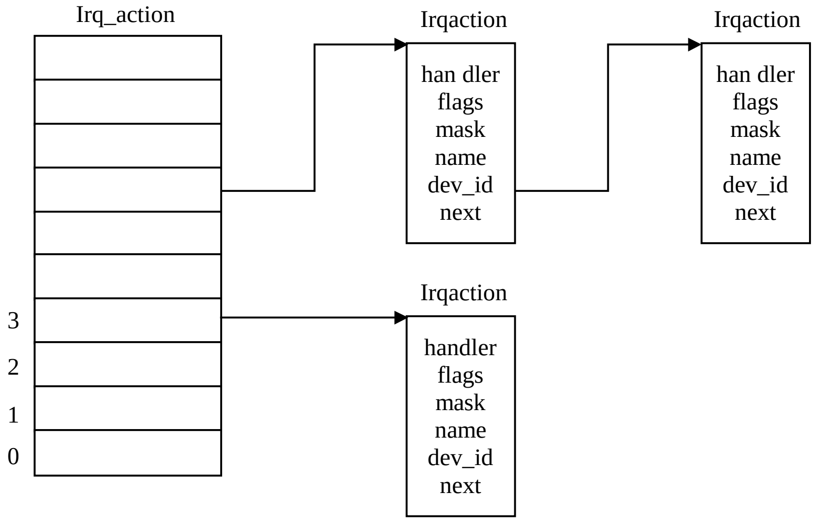


图 11.5　irqaction 的数据结构

根据设备的中断号可以在数组 irq_action 检索到设备的中断信息。对中断资源的请求在驱动程序初始化时就已经完成。

在传统的 PC 体系结构中 ，有些中断已经被固定下来。软盘设备正是这种情况，它的中断号总为 6。有时设备驱动程序可能不知道设备使用的中断号，对 PCI 设备来说这不是什么大问题，它们总是可以通过设备配置头知道其中断号。但对于 ISA 设备则没有取得中断号的方便方式，Linux 通过让设备驱动程序检测它们的中断号来解决这个问题。

让我们来看一下对 ISA 设备中断号的检测过程。设备驱动程序首先迫使 ISA 设备引起一个中断，系统中所有未被分配的中断都被打开。此时设备引发的中断可以通过可编程中断控制器来发送出去，在它接受到 CPU 的响应信号以后将中断号放置在数据线上，Linux 读取此数据并将其内容返回给设备驱动程序。非 0 结果则表示在此次检测中有中断发生，设备驱动程序然后将关闭检测并将所有未分配中断屏蔽掉，这样 ISA 设备驱动程序就成功地找到了设备的 IRQ 号。

基于 PCI 系统比基于 ISA 系统有更多的动态性。ISA 设备使用的中断引脚通常是通过硬件设备上的跳线来设置的。而每个 PCI 设备都对应一个配置头， PCI 设备在系统启动与初始化 PCI 时由 PCI BIOS 或 PCI 子系统来分配中断，将其放入配置头中，故而驱动程序可以方便获得 PCI 设备使用的中断号。

系统中可能存在许多 PCI 中断源，比如在使用 PCI-PCI 桥接器时。这些中断源的个数可能将超出系统可编程中断控制器的引脚数。此时 PCI 设备必须共享中断号，中断控制器上的一个引脚可能被多个 PCI 设备同时使用。Linux 让中断的第 1 个请求者申明此中断是否可以共享，中断的共享将导致 irq_action 数组中的一个入口同时指向几个 irqaction 数据结构，如图 11.5 所示。当共享中断发生时 Linux 将调用对应此中断源的所有中断处理过程。

2．Linux 对中断的处理

Linux 中断处理子系统的一个基本任务是将中断正确联系到中断处理代码中的正确位

置。这些代码必须了解系统的中断拓扑结构。例如在中断控制器上引脚 6 上发生的软盘控制器中断必须被辨认出的确来自软盘并同系统的软盘设备驱动的中断服务子程序联系起来。

中断发生时，Linux 首先读取系统可编程中断控制器中中断状态寄存器，判断出中断源，将其转换成 irq_action 数组中偏移值（例如来自软盘控制器引脚 6 的中断将被转换成对应于 irq_action 数组中的第 7 个指针），然后调用其相应的中断处理程序。

当 Linux 内核调用设备驱动程序的中断服务子程序时，必须找出中断产生的原因以及相应的解决办法，这是通过读取设备上的状态寄存器的内容来完成的。

下面我们结合输入/输出系统的层次结构来看一下中断在驱动程序工作的过程中的作用。

（1）用户发出某种输入/输出请求。

（2）调用驱动程序的 read() 函数或 request() 函数，将完成的输入/输出的指令送给设备控制器，现在设备驱动程序等待操作的发生。

（3）一小段时间以后，硬设备准备好完成指令的操作，并产生中断信号标志事件的发生。

（4）中断信号导致调用驱动程序的中断服务子程序，它将所要的数据从硬设备复制到设备驱动程序的缓冲区中，并通知正在等待的 read() 函数和 request()函数，现在数据可供使用。

（5）在数据可供使用时，read()或 request()函数现在可将数据提供给用户进程。

上述过程是经过了简化了的，但却反映了中断的主要过程的主要方面。

11.2.6 驱动 DMA 工作

所有的 PC 都包含一个称为直接内存访问控制器或 DMAC 的辅助处理器，它可以用来控制在 RAM 和 I/O 设备之间数据的传送。DMAC 一旦被 CPU 激活，就可以自行传送数据；当数据传送完成之后，DMAC 发出一个中断请求。当 CPU 和 DMAC 同时访问同一内存单元时，所产生的冲突由一个称为内存仲裁器的硬件电路来解决。

使用 DMAC 最多的是磁盘驱动器和其他需要一次传送大量字节的慢速设备。因为 DMAC 的设置时间相当长，所以在传送数量很少的数据时直接使用 CPU 效率更高。

原来的 ISA 总线所使用的第一个 DMAC 非常复杂，难于对其进行编程。PCI 和 SCSI 总线所使用的最新 DMAC 依靠总线中的专用硬件电路，这就使设备驱动程序开发人员的开发工作变得简单。

到现在为止，我们已区分了 3 类内存地址：逻辑地址、线性地址以及物理地址，前两个在 CPU 内部使用，最后一个是 CPU 从物理上驱动数据总线所用的内存地址。但是，还有第 4 种内存地址，称为总线地址：它是除 CPU 之外的硬件设备驱动数据总线所用的内存地址。在 PC 体系结构中，总线地址和物理地址是一致的；但是在其他体系结构中，例如 Sun 的 SPARC 和 Compaq 的 Alpha 体系结构中，这两种地址是不同的。

从根本上说，内核为什么应该关心总线地址呢？这是因为在 DMA 操作中数据传送不用 CPU 的参与：I/O 设备和 DMAC 直接驱动数据总线。因此，在内核开始 DMA 操作时，必须把所涉及的内存缓冲区总线地址或写入 DMAC 适当的 I/O 端口、或写入 I/O 设备适当的 I/O 端口。

很多 I/O 驱动程序都使用直接内存访问控制器（DMAC）来加快操作的速度。DMAC 与设备的 I/O 控制器相互作用共同实现数据传送。后文中我们还会看到，内核中包含一组易用的例程来对 DMAC 进行编程。当数据传送完成时，I/O 控制器通过 IRQ 向 CPU 发出信号。

当设备驱动程序为某个 I/O 设备建立 DMA 操作时，必须使用总线地址指定所用的内存缓冲区。内核提供两个宏 virt_to_bus 和 bus_to_virt，分别把虚拟地址转换成总线地址或把总线地址转换成虚拟地址。

与 IRQ 一样，DMAC 也是一种资源，必须把这种资源动态地分配给需要它的设备驱动程序。驱动程序开始和结束 DMA 操作的方法依赖于总线的类型。

1．ISA 总线的 DMA

每个 ISA DMAC 只能控制有限个通道。每个通道都包括一组独立的内部寄存器，所以，DMAC 就可以同时控制几个数据的传送。

设备驱动程序通常使用下面的方式来申请和释放 ISA DMAC。设备驱动程序照样要靠一个引用计数器来检测什么时候任何进程都不再访问设备文件。驱动程序执行以下操作。

（1）在设备文件的 open( )方法中把设备的引用计数器加 1。如果原来的值是 0，那么，驱动程序执行以下操作：

- 调用 request_irq( )来分配 ISA DMAC 所使用的 IRQ 中断号；
- 调用 request_dma( )来分配 DMA 通道；
- 通知硬件设备应该使用 DMA 并产生中断。
- 如果需要，为 DMA 缓冲区分配一个存储区域

（2）当必须启动 DMA 操作时，在设备文件的 read（）和 write（）方法中执行以下操作：

- 调用 set_dma_mode( )把通道设置成读/写模式；
- 调用 set_dma_addr( )来设置 DMA 缓冲区的总线地址。(因为只有最低的 24 位地址会发给 DMAC，所以缓冲区必须在 RAM 的前 16MB 中)；
- 调用 set_dma_count( )来设置要发送的字节数；
- 调用 set_dma_dma( )来启用 DMA 通道；
- 把当前进程加入该设备的等待队列，并把它挂起，当 DMAC 完成数据传送操作时，设备的 I/O 控制器就发出一个中断，相应的中断处理程序会唤醒正在睡眠的进程；
- 进程一旦被唤醒，就立即调用 disable_dma( )来禁用这个 DMA 通道；
- 调用 get_dma_residue( )来检查是否所有的数据都已被传送。

（3）在设备文件的 release 方法中，减少设备的引用计数器。如果该值变成 0，就执行以下操作：

- 禁用 DMA 和对这个硬件设备上的相应中断；
- 调用 free_dma( )来释放 DMA 通道；
- 调用 free_irq( )来释放 DMA 所使用的 IRQ 线。

2．PCI 总线的 DMA

PCI 总线对于 DMA 的使用要简单得多，因为 DMAC 是集成到 I/O 接口内部的。在 open()方法中，设备驱动程序照样必须分配一条 IRQ 线来通知 DMA 操作的完成。但是，并没有必要

分配一个 DMA 通道，因为每个硬件设备都直接控制 PCI 总线的电信号。要启动 DMA 操作，设备驱动程序在硬件设备的某个 I/O 端口中简单地写入 DMA 缓冲区的总线地址、传送方向以及数据大小，然后驱动程序就挂起当前进程。在最后一个进程关闭这个文件对象时，release 方法负责释放这条 IRQ 线。

11.2.7 I/O 空间的映射

很多硬件设备都有自己的内存，通常称之为 I/O 空间。例如，所有比较新的图形卡都有几 MB 的 RAM，称为显存，用它来存放要在屏幕上显示的屏幕影像。

1．地址映射

根据设备和总线类型的不同，PC 体系结构中的 I/O 空间可以在 3 个不同的物理地址范围之间进行映射。

（1）对于连接到 ISA 总线上的大多数设备

I/O 空间通常被映射到从 0xa0000 到 0xfffff 的物理地址范围，这就在 640K 和 1MB 之间留出了一段空间，这就是所谓的“洞”。

（2）对于使用 VESA 本地总线（VLB）的一些老设备

这主要是由图形卡使用的一条专用总线：I/O 空间被映射到从 0xe00000 到 0xffffff 的地址范围中，也就是 14MB 到 16MB 之间。因为这些设备使页表的初始化更加复杂，因此已经不生产这种设备了。

（3）对于连接到 PCI 总线的设备

I/O 空间被映射到很大的物理地址区间，位于 RAM 物理地址的顶端。这种设备的处理比较简单。

2．访问 I/O 空间

内核如何访问一个 I/O 空间单元？让我们从 PC 体系结构开始入手，这个问题很容易就可以解决，之后我们再进一步讨论其他体系结构。

不要忘了内核程序作用于虚拟地址，因此 I/O 空间单元必须表示成大于 PAGE_OFFSET 的地址。在后面的讨论中，我们假设 PAGE_OFFSET 等于 0xc0000000，也就是说，内核虚拟地址是在第 4GB。

内核驱动程序必须把 I/O 空间单元的物理地址转换成内核空间的虚拟地址。在 PC 体系结构中，这可以简单地把 32 位的物理地址和 0xc0000000 常量进行或运算得到。例如，假设内核需要把物理地址为 0x000b0fe4 的 I/O 单元的值存放在 t1 中，把物理地址为 0xfc000000 的 I/O 单元的值存放在 t2 中，就可以使用下面的表达式来完成这项功能：

```
t1 = *((unsigned char *)(0xc00b0fe4));
t2 = *((unsigned char *)(0xfc000000));
```

在第六章我们已经介绍过，在初始化阶段，内核已经把可用的 RAM 物理地址映射到虚拟地址空间第 4GB 的最初部分。因此，分页机制把出现在第 1 个语句中的虚拟地址 0xc00b0fe4 映射回到原来的 I/O 物理地址 0x000b0fe4，正好落在从 640K 到 1MB 的这段“ISA 洞”中。这

正是我们所期望的。

但是，对于第 2 个语句来说，这里有一个问题，因为其 I/O 物理地址超过了系统 RAM 的最大物理地址。因此，虚拟地址 0xfc000000 就不需要与物理地址 0xfc000000 相对应。在这种情况下，为了在内核页表中包括对这个 I/O 物理地址进行映射的虚拟地址，必须对页表进行修改：这可以通过调用 ioremap( )函数来实现。ioremap( )和 vmalloc( )函数类似，都调用 get_vm_area( ) 建立一个新的 vm_struct 描述符，其描述的虚拟地址区间为所请求 I/O 空间区的大小。然后，ioremap( )函数适当地更新所有进程的对应页表项。

因此，第 2 个语句的正确形式应该为：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

第 1 条语句建立一个 2MB 的虚拟地址区间，从 0xfb000000 开始；第 2 条语句读取地址 0xfc000000 的内存单元。驱动程序以后要取消这种映射，就必须使用 iounmap( )函数。

现在让我们考虑一下除 PC 之外的体系结构。在这种情况下，把 I/O 物理地址加上 0xc0000000 常量所得到的相应虚拟地址并不总是正确的。为了提高内核的可移植性，Linux 特意包含了下面这些宏来访问 I/O 空间。

```
readb, readw, readl
```

分别从一个 I/O 空间单元读取 1、2 或者 4 个字节。

```
writeb, writew, writel
```

分别向一个 I/O 空间单元写入 1、2 或者 4 个字节。

```
memcpy_fromio, memcpy_toio
```

把一个数据块从一个 I/O 空间单元拷贝到动态内存中，另一个函数正好相反，把一个数据块从动态内存中拷贝到一个 I/O 空间单元。

```
memset_io
```

用一个固定的值填充一个 I/O 空间区域。

对于 0xfc000000 I/O 单元的访问推荐使用如下方法：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

使用这些宏，就可以隐藏不同平台访问 I/O 空间所用方法的差异。

11.2.8　设备驱动程序框架

由于设备种类繁多，相应的设备驱动程序也非常之多。尽管设备驱动程序是内核的一部分，但设备驱动程序的开发往往由很多人来完成，如业余编程高手、设备厂商等。为了让设备驱动程序的开发建立在规范的基础上，就必须在驱动程序和内核之间有一个严格定义和管理的接口，例如 SVR4 提出了 DDI/DDK 规范，其含义就是设备与驱动程序接口／设备驱动程序与内核接口（Device-Driver Interface／Driver-Kernel Interface)。通过这个规范，可以规范设备驱动程序与内核之间的接口。

Linux 的设备驱动程序与外接的接口与 DDI/DKI 规范相似，可以分为以下 3 部分。

（1）驱动程序与内核的接口，这是通过数据结构 file_operations 来完成的。

（2）驱动程序与系统引导的接口，这部分利用驱动程序对设备进行初始化。

（3）驱动程序与设备的接口，这部分描述了驱动程序如何与设备进行交互，这与具体设

备密切相关。

根据功能，驱动程序的代码可以分为如下几个部分。

（1）驱动程序的注册和注销。

（2）设备的打开与释放。

（3）设备的读和写操作。

（4）设备的控制操作。

（5）设备的中断和查询处理。

前三点我们已经给予了简单说明，后面我们还会结合具体程序给出进一步的说明。关于设备的控制操作可以通过驱动程序中的 ioctl()来完成,例如,对光驱的控制可以使用 cdrom_ioctl()。

与读写操作不同，ioctl()的用法与具体设备密切相关，例如，对于软驱的控制可以使用 floppy_ioctl()，其调用形式为：

```
static int floppy_ioctl(struct inode *inode, struct file *filp,
                        unsigned int cmd, unsigned long param)
```

其中 cmd 的取值及含义与软驱有关，例如，FDEJECT 表示弹出软盘。

除了 ioctl()，设备驱动程序还可能有其他控制函数，如 lseek（）等。

对于不支持中断的设备，读写时需要轮流查询设备的状态，以便决定是否继续进行数据传输，例如，打印机驱动程序在缺省时轮流查询打印机的状态。

如果设备支持中断，则可按中断方式进行。

11.3　块设备驱动程序

对于块设备来说，读写操作是以数据块为单位进行的，为了使高速的 CPU 同低速块设备能够协调工作，提高读写效率，操作系统设置了缓冲机制。当进行读写的时候，首先对缓冲区读写，只有缓冲区中没有需要读的数据或是需要写的数据没有地方写时，才真正地启动设备控制器去控制设备本身进行数据交换，而对于设备本身的数据交换同样也是同缓冲区打交道。

11.3.1　块设备驱动程序的注册

对于块设备来说，驱动程序的注册不仅在其初始化的时候进行而且在编译的时候也要进行注册。在初始化时通过 register_blkdev( ) 函数将相应的块设备添加到数组 blkdevs 中，该数组在 fs/block_dev.c 中定义如下：

```
static struct {
        const char *name;
        struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];
```

从 Linux 2.4 开始，块设备表的定义与下一节要介绍的字符设备表的定义有所不同。因为每种具体的块设备都有一套具体的操作，因而各自有一个类似于 file_operations 那样的

数据结构，称为 block_device_operations 结构，其定义为：

```
struct block_device_operations {
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
        int (*check_media_change) (kdev_t);
        int (*revalidate) (kdev_t);
         struct module *owner;
};
```

如果说 file_operation 结构是连接虚拟的 VFS 文件的操作与具体文件系统的文件操作之间的枢纽，那么 block_device_operations 就是连接抽象的块设备操作与具体块设备操作之间的枢纽。

具体的块设备是由主设备号唯一确定的，因此，主设备号唯一地确定了一个具体的 block_device_operations 数据结构。

下面我们来看 register_blkdev（ ）函数的具体实现，其代码在 fs/block_dev.c 中：

```
int register_blkdev(unsigned int major, const char * name, struct block_device_operations *bdops)
{
         if (major == 0) {
                 for (major = MAX_BLKDEV-1; major > 0; major--) {
                         if (blkdevs[major].bdops == NULL) {
                                 blkdevs[major].name = name;
                                 blkdevs[major].bdops = bdops;
                                 return major;
                         }
                 }
                return -EBUSY;
         }
         if (major >= MAX_BLKDEV)
                 return -EINVAL;
        if (blkdevs[major].bdops && blkdevs[major].bdops != bdops)
                 return -EBUSY;
         blkdevs[major].name = name;
         blkdevs[major].bdops = bdops;
        return 0;
}
```

这个函数的第 1 个参数是主设备号，第 2 个参数是设备名称的字符串，第 3 个参数是指向具体设备操作的指针。如果一切顺利则返回 0，否则返回负值。如果指定的主设备号为 0，此函数将会搜索空闲的主设备号分配给该设备驱动程序并将其作为返回值。

那么，块设备注册到系统以后，怎样与文件系统联系起来呢，也就是说，文件系统怎么调用已注册的块设备，这还得从 file_operations 结构说起。

我们先来看一下块设备的 file_operations 结构的定义，其位于 fs/block_dev.c 中：

```
struct file_operations def_blk_fops = {
        open:           blkdev_open,
        release:        blkdev_close,
        llseek:         block_llseek,
         read:           generic_file_read,
```

```
        write:          generic_file_write,
        mmap:           generic_file_mmap,
        fsync:          block_fsync,
        ioctl:          blkdev_ioctl,
};
```

下面以 open()系统调用为例，说明用户进程中的一个系统调用如何最终与物理块设备的操作联系起来。在此，我们仅仅给出几个 open（）函数的调用关系，如图 11.6 所示。

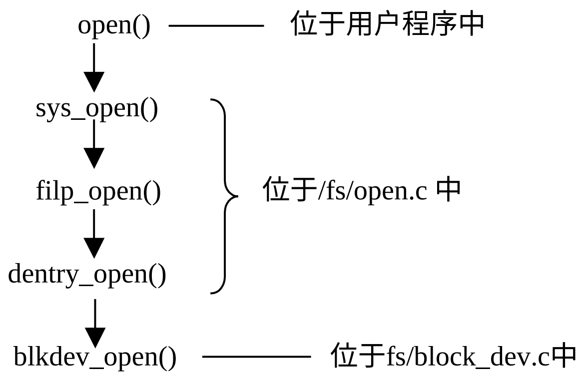


图 11.6　几个 open()函数的调用关系

如图 11.6 所示，当调用 open（）系统调用时，其最终会调用到 def_blk_fops 的 blkdev_open（）函数。blkdev_open（）函数的任务就是根据主设备号找到对应的 block_device_operations 结构，然后再调用 block_device_operations 结构中的函数指针 open 所指向的函数，如果 open 所指向的函数非空，就调用该函数打开最终的物理块设备。这就简单地说明了块设备注册以后，从最上层的系统调用到具体地打开一个设备的过程。

另外要说明的是，如果选择了通过设备文件系统 DevFS 进行注册，则调用 devfs_register_blkdev()函数，该函数的说明及代码在 fs/devfs/base.c 中定义如下：

```
/**
*      devfs_register_blkdev - Optionally register a conventional block driver.
  *      @major: The major number for the driver.
  *      @name: The name of the driver (as seen in /proc/devices).
  *      @bdops: The &block_device_operations structure pointer.
  *
  *      This function will register a block driver provided the "devfs=only"
  *      option was not provided at boot time.
  *      Returns 0 on success, else a negative error code on failure.
  */

int devfs_register_blkdev (unsigned int major, const char *name,
                            struct block_device_operations *bdops)
{
     if (boot_options & OPTION_ONLY) return 0;
     return register_blkdev (major, name, bdops);
}   /*  End Function devfs_register_blkdev  */
```

11.3.2　块设备基于缓冲区的数据交换

关于块缓冲区的管理在第八章虚拟文件系统中已有所描述，在这里我们从交换数据的角度来看一下基于缓冲区的数据交换的实现。

1．扇区及块缓冲区

块设备的每次数据传送操作都作用于一组相邻字节，我们称之为扇区。在大部分磁盘设备中，扇区的大小是 512 字节，但是现在新出现的一些设备使用更大的扇区（1024 和 2014 字节）。注意，应该把扇区作为数据传送的基本单元：不允许传送少于一个扇区的数据，而大部分磁盘设备都可以同时传送几个相邻的扇区。

所谓块就是块设备驱动程序在一次单独操作中所传送的一大块相邻字节。注意不要混淆块（block）和扇区（sector）：扇区是硬件设备传送数据的基本单元，而块只是硬件设备请求一次 I/O 操作所涉及的一组相邻字节。

在 Linux 中，块大小必须是 2 的幂，而且不能超过一个页面。此外，它必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此，在 PC 体系结构中，允许块的大小为 512、1024、2048 和 4096 字节。同一个块设备驱动程序可以作用于多个块大小，因为它必须处理共享同一主设备号的一组设备文件，而每个块设备文件都有自己预定义的块大小。例如，一个块设备驱动程序可能会处理有两个分区的硬盘，一个分区包含 Ext2 文件系统，另一个分区包含交换分区。

内核在一个名为 blksize_size 的表中存放块的大小；表中每个元素的索引就是相应块设备文件的主设备号和从设备号。如果 blksize_size[M]为 NULL，那么共享主设备号 M 的所有块设备都使用标准的块大小，即 1024 字节。

每个块都需要自己的缓冲区，它是内核用来存放块内容的 RAM 内存区。当设备驱动程序从磁盘读出一个块时，就用从硬件设备中所获得的值来填充相应的缓冲区；同样，当设备驱动程序向磁盘中写入一个块时，就用相关缓冲区的实际值来更新硬件设备上相应的一组相邻字节。缓冲区的大小一定要与块的大小相匹配。

2．块驱动程序的体系结构

下面我们说明通用块驱动程序的体系结构，以及在为缓冲区 I/O 操作时所涉及的主要成分。

块设备驱动程序通常分为两部分，即高级驱动程序和低级驱动程序，前者处理 VFS 层，后者处理硬件设备，如图 11.7 所示。

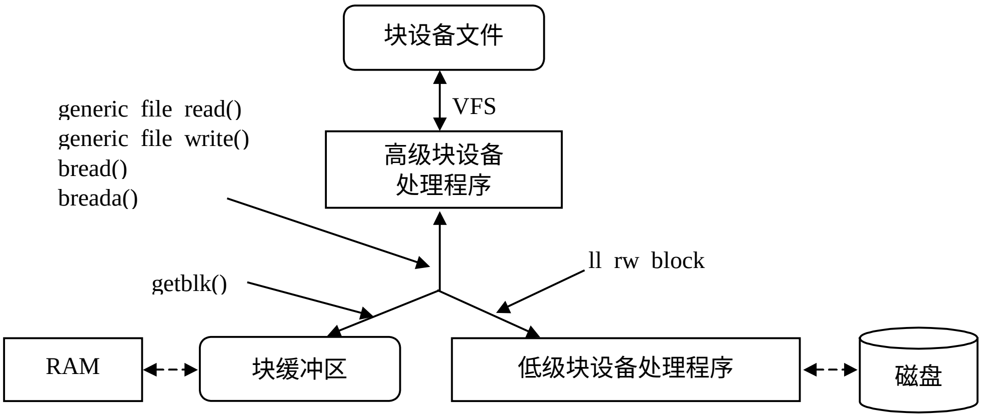


图 11.7　块设备驱动程序的体系结构

假设进程对一个设备文件发出 read( )或 write( )系统调用。VFS 执行对应文件对象的 read 或 write 方法，由此就调用高级块设备处理程序中的一个过程。这个过程执行的所有操作都与对这个硬件设备的具体读写请求有关。内核提供两个名为 generic_file_read（)和 generic_file_write（)通用函数来留意所有事件的发生。因此，在大部分情况下，高级硬件设备驱动程序不必做什么，而设备文件的 read 和 write 方法分别指向 generic_file_read（)和 generic_file_write（)方法。

但是，有些块设备的处理程序需要自己专用的高级设备驱动程序。典型的例子是软驱的设备驱动程序：它必须检查从上次访问磁盘以来，用户有没有改变驱动器中的磁盘；如果已插入一张新磁盘，那么设备驱动程序必须使缓冲区中所包含的旧数据无效。

即使高级设备驱动程序有自己的 read 和 write 方法，但是这两个方法通常最终还会调用 generic_file_read（)和 generic_file_write（)函数。这些函数把对 I/O 设备文件的访问请求转换成对相应硬件设备的块请求。所请求的块可能已在主存，因此 generic_file_read（)和 generic_file_write（)函数调用 getblk( )函数来检查缓冲区中是否已经预取了块，还是从上次访问以来缓冲区一直都没有改变。如果块不在缓冲区中，getblk( )就必须调用 ll_rw_block( )继续从磁盘中读取这个块，后面这个函数激活操纵设备控制器的低级驱动程序，以执行对块设备所请求的操作。

在 VFS 直接访问某一块设备上的特定块时，也会触发缓冲区 I/O 操作。例如，如果内核必须从磁盘文件系统中读取一个索引节点，那么它必须从相应磁盘分区的块中传送数据 。对于特定块的直接访问是由 bread( )和 breada( )函数来执行的，这两个函数又会调用前面提到过的 getblk( )和 ll_rw_block( )函数。

由于块设备速度很慢，因此缓冲区 I/O 数据传送通常都是异步处理的：低级设备驱动程序对 DMAC 和磁盘控制器进行编程来控制其操作，然后结束。当数据传送完成时，就会产生一个中断，从而第 2 次激活这个低级设备驱动程序来清除这次 I/O 操作所涉及的数据结构。

3．块设备请求

虽然块设备驱动程序可以一次传送一个单独的数据块，但是内核并不会为磁盘上每个被访问的数据块都单独执行一次 I/O 操作：这会导致磁盘性能的下降，因为确定磁盘表面块的

物理位置是相当费时的。取而代之的是，只要可能，内核就试图把几个块合并在一起，并作为一个整体来处理，这样就减少了磁头的平均移动时间。

当进程、VFS 层或者任何其他的内核部分要读写一个磁盘块时，就真正引起一个块设备请求。从本质上说，这个请求描述的是所请求的块以及要对它执行的操作类型（读还是写）。然而，并不是请求一发出，内核就满足它，实际上，块请求发出时 I/O 操作仅仅被调度，稍后才会被执行。这种人为的延迟有悖于提高块设备性能的关键机制。当请求传送一个新的数据块时，内核检查能否通过稍微扩大前一个一直处于等待状态的请求而满足这个新请求，也就是说，能否不用进一步的搜索操作就能满足新请求。由于磁盘的访问大都是顺序的，因此这种简单机制就非常高效。

延迟请求复杂化了块设备的处理。例如，假设某个进程打开了一个普通文件，然后，文件系统的驱动程序就要从磁盘读取相应的索引节点。高级块设备驱动程序把这个请求加入一个等待队列，并把这个进程挂起，直到存放索引节点的块被传送为止。

因为块设备驱动程序是中断驱动的，因此，只要高级驱动程序一发出块请求，它就可以终止执行。在稍后的时间低级驱动程序才被激活，它会调用一个所谓的策略程序从一个队列中取得这个请求，并向磁盘控制器发出适当的命令来满足这个请求。当 I/O 操作完成时，磁盘控制器就产生一个中断，如果需要，相应的处理程序会再次调用这个策略程序来处理队列中进程的下一个请求。

每个块设备驱动程序都维护自己的请求队列；每个物理块设备都应该有一个请求队列，以提高磁盘性能的方式对请求进行排序。因此策略程序就可以顺序扫描这种队列，并以最少地移动磁头而为所有的请求提供服务。

每个块设备请求都是由一个 request 结构来描述的，其定义于 include/linux/blkdev.h：

```
/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests.
 */
struct request {
        struct list_head queue;
        int elevator_sequence;

        volatile int rq_status; /* should split this into a few status bits */
#define RQ_INACTIVE             (-1)
#define RQ_ACTIVE               1
#define RQ_SCSI_BUSY            0xffff
#define RQ_SCSI_DONE            0xfffe
#define RQ_SCSI_DISCONNECTING   0xffe0

        kdev_t rq_dev;
        int cmd;                /* READ or WRITE */
        int errors;
        unsigned long sector;
        unsigned long nr_sectors;
        unsigned long hard_sector, hard_nr_sectors;
        unsigned int nr_segments;
        unsigned int nr_hw_segments;
        unsigned long current_nr_sectors;
```

```
            void * special;
             char * buffer;
             struct completion * waiting;
             struct buffer_head * bh;
             struct buffer_head * bhtail;
             request_queue_t *q;
    };
```

从代码注释可以知道，在 2.2 以前的版本中没有这么多域，很多域是为分页请求而增加的，我们暂且不予考虑。在此，我们只说明与块传送有关的域。为了描述方便起见，我们把 struct request 叫做请求描述符。

数据传送的方向存放在 cmd 域中：该值可能是 READ（把数据从块设备读到 RAM 中）或者 WRITE（把数据从 RAM 写到块设备中）。rq_status 域用来定义请求的状态：对于大部分块设备来说，这个域的值可能为 RQ_INACTIVE（请求描述符还没有使用）或者 RQ_ACTIVE（有效的请求，低级设备驱动程序要对其服务或正在对其服务）。

一次请求可能包括同一设备中的很多相邻块。rq_dev 域指定块设备，而 sector 域说明请求中第一个块对应的第一个扇区的编号。nr_sector 和 current_nr_sector 给出要传送数据的扇区数。sector、nr_sector 和 current_nr_sector 域都可以在请求得到服务的过程中而被动态修改。

请求块的所有缓冲区首部都被集中在一个简单链表中。每个缓冲区首部的 b_reqnext 域指向链表中的下一个元素，而请求描述符的 bh 和 bhtail 域分别指向链表的第一个元素和最后一个元素。

请求描述符的 buffer 域指向实际数据传送所使用的内存区。如果只请求一个单独的块，那么缓冲区只是缓冲区首部的 b_data 域的一个拷贝。然而，如果请求了多个块，而这些块的缓冲区在内存中又不是连续的，那么就使用缓冲区首部的 b_reqnext 域把这些缓冲区链接在一起。对于读操作来说，低级设备驱动程序可以选择先分配一个大的内存区来立即读取请求的所有扇区，然后再把这些数据拷贝到各个缓冲区。同样，对于写操作来说，低级设备驱动程序可以把很多不连续缓冲区中的数据拷贝到一个单独内存区的缓冲区中，然后再立即执行整个数据的传送。

另外，在严重负载和磁盘操作频繁的情况下，固定数目的请求描述符就可能成为一个瓶颈。空闲描述符的缺乏可能会强制进程等待直到正在执行的数据传送结束。因此，request_queue_t 类型（见下面）中的 wait_for_request 等待队列就用来对正在等待空闲请求描述符的进程进行排队。get_request_wait( )试图获取一个空闲的请求描述符，如果没有找到，就让当前进程在等待队列中睡眠；get_request( )函数与之类似，但是如果没有可用的空闲请求描述符，它只是简单地返回 NULL。

4．请求队列

请求队列只是一个简单的链表，其元素是请求描述符。每个请求描述符中的 next 域都指向请求队列的下一个元素，最后一个元素为空。这个链表的排序通常是：首先根据设备标识符，其次根据最初的扇区号。

如前所述，对于所服务的每个硬盘，设备驱动程序通常都有一个请求队列。然而，一些

设备驱动程序只有一个请求队列，其中包括了由这个驱动器处理的所有物理设备的请求。这种方法简化了驱动程序的设计，但是损失了系统的整体性能，因为不能对队列强制使用简单排序的策略。请求队列定义如下：

```
struct request_queue
{
        /*
         * the queue request freelist, one for reads and one for writes
         */
        struct request_list     rq[2];

        /*
         * Together with queue_head for cacheline sharing
         */
        struct list_head        queue_head;
        elevator_t              elevator;

        request_fn_proc         * request_fn;
        merge_request_fn        * back_merge_fn;
        merge_request_fn        * front_merge_fn;
        merge_requests_fn       * merge_requests_fn;
        make_request_fn         * make_request_fn;
        plug_device_fn          * plug_device_fn;
        /*
         * The queue owner gets to use this for whatever they like.
         * ll_rw_blk doesn't touch it.
         */
        void                    * queuedata;

        /*
         * This is used to remove the plug when tq_disk runs.
         */
        struct tq_struct        plug_tq;

        /*
         * Boolean that indicates whether this queue is plugged or not.
         */
        char                    plugged;

        /*
         * Boolean that indicates whether current_request is active or
         * not.
         */
        char                    head_active;

        /*
         * Is meant to protect the queue in the future instead of
         * io_request_lock
         */
        spinlock_t              queue_lock;

        /*
```

```
             * Tasks wait here for free request
             */
            wait_queue_head_t       wait_for_request;
    };
typedef struct request_queue request_queue_t;
```

其中，request_list 为请求描述符组成的空闲链表，其定义如下：

```
  struct request_list {
        unsigned int count;
         struct list_head free;
  };
```

有两个这样的链表，一个用于读，一个用于写。

elevator_t 结构描述的是为磁盘的电梯调度算法而设的数据结构。从 request_fn_proc 到 plug_device_fn 都是一些函数指针。例如 request_fn 是一个指针，指向类型为 request_fn_proc 的对象。而 request_fn_proc 则通过#typedef 定义为一种函数：

```
typedef void (request_fn_proc) (request_queue_t *q)
```

其余的函数也与此类似，这些指针（连同其他域）都是在相应设备初始化时设置好的。需要对一个块设备进行操作时，就为之设置好一个数据结构 request_queue。并将其挂入相应的请求队列中。

这里要说明的是，request_fn( )域包含驱动程序的策略程序的地址，策略程序是低级块设备驱动程序的关键函数，为了开始传送队列中的一个请求所指定的数据，它与物理块设备（通常是磁盘控制器）真正打交道。

5．块设备驱动程序描述符

驱动程序描述符是一个 blk_dev_struct 类型的数据结构，其定义如下：

```
   struct blk_dev_struct {
        /*
        * queue_proc has to be atomic
        */
         request_queue_t         request_queue;
         queue_proc              *queue;
         void                    *data;
   };
```

在这个结构中，其主体是请求队列 request_queue；此外，还有一个函数指针 queue，当这个指针为非 0 时，就调用这个函数来找到具体设备的请求队列，这是为考虑具有同一主设备号的多种同类设备而设的一个域。这个指针也在设备初始化时就设置好，另一个指针 data 是辅助 queue 函数找到特定设备的请求队列。

所有块设备的描述符都存放在 blk_dev 表中：

```
struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

每个块设备都对应着数组中的一项，可以用主设备号进行检索。每当用户进程对一个块设备发出一个读写请求时，首先调用块设备所公用的函数 generic_file_read ( )和 generic_file_write ( )，如果数据存在缓冲区中或缓冲区还可以存放数据，就同缓冲区进行数据交换。否则，系统会将相应的请求队列结构添加到其对应项的 blk_dev_struct 中，如图 11.8 所示。如果在加入请求队列结构的时候该设备没有请求，则马上响应该请求，否则

将其追加到请求任务队列尾顺序执行。

图 11.8 表示每个请求有指向一个或多个 buffer_hear 结构的指针，每个请求读写一块数据。如果系统对 buffer_head 结构上锁，则进程会等待到对此缓冲区的块操作完成。一旦设备驱动程序完成了请求则它必须将每个 buffer_heard 结构从 request 结构中清除，将它们标记成已更新状态并对它们解锁。对 buffer_head 的解锁将唤醒所有等待此块操作完成的睡眠进程，然后 request 数据结构被标记成空闲以便被其他块请求使用。

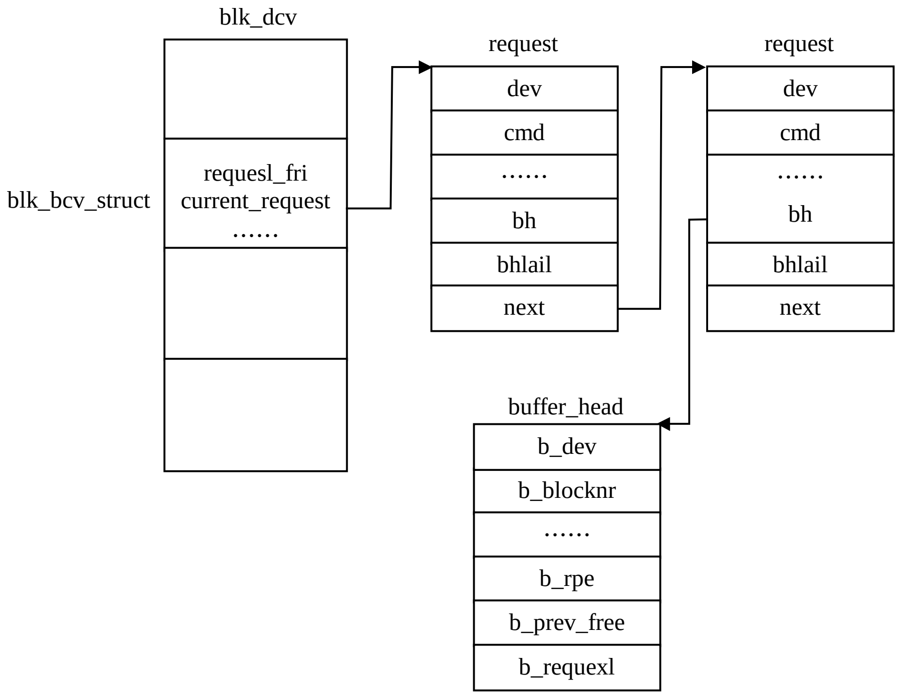


图 11.8　块设备读写请求

11.3.3　块设备驱动程序的几个函数

所有对块设备的读写都是调用 generic_file_read（ ）和 generic_file_write（ ）函数，这两个函数的原型如下：

```
ssize_t generic_file_read(struct file * filp, char * buf, size_t count, loff_t   *ppos)
ssize_t   generic_file_write(struct file *file,const char *buf,size_t count, loff_t  *ppos)
```

其参数的含义如下。

filp：和这个设备文件相对应的文件对象的地址。

Buf：用户态地址空间中的缓冲区的地址。generic_file_read（）把从块设备中读出的数据写入这个缓冲区；反之，generic_file_write（）从这个缓冲区中读取要写入块设备的数据。

Count：要传送的字节数。

ppos：设备文件中的偏移变量的地址；通常，这个参数指向 filp->f_pos，也就是说，

指向设备文件的文件指针。

只要进程对设备文件发出读写操作，高级设备驱动程序就调用这两个函数。例如，superformat 程序通过把块写入/dev/fd0 设备文件来格式化磁盘，相应文件对象的 write 方法就调用 generic_file_write（）函数。这两个函数所做的就是对缓冲区进行读写，如果缓冲区不能满足操作要求则返回负值，否则返回实际读写的字节数。每个块设备在需要读写时都调用这两个函数。

下面介绍几个低层被频繁调用的函数。

bread( )和 breada( ) 函数

bread( )函数检查缓冲区中是否已经包含了一个特定的块；如果还没有，该函数就从块设备中读取这个块。文件系统广泛使用 bread( )从磁盘位图、索引节点以及其他基于块的数据结构中读取数据（注意当进程要读块设备文件时是使用 generic_file_read（）函数，而不是使用 bread( )函数）。该函数接收设备标志符、块号和块大小作为参数，其代码在 fs/buffer.c 中：

```
/**
 *      bread() - reads a specified block and returns the bh
 *      @block: number of block
 *      @size: size (in bytes) to read
 *
  *      Reads a specified block, and returns buffer head that
  *      contains it. It returns NULL if the block was unreadable.
  */
struct buffer_head * bread(kdev_t dev, int block, int size)
{
        struct buffer_head * bh;

       bh = getblk(dev, block, size);
       touch_buffer(bh);
       if (buffer_uptodate(bh))
               return bh;
        ll_rw_block(READ, 1, &bh);
         wait_on_buffer(bh);
         if (buffer_uptodate(bh))
                return bh;
        brelse(bh);
        return NULL;
}
```

对该函数解释如下。

- 调用 getblk( )函数来查找缓冲区中的一个块；如果这个块不在缓冲区中，那么 getblk( )就为它分配一个新的缓冲区。
- 调用 buffer_uptodate（）宏来判断这个缓冲区是否已经包含最新数据，如果是，则 getblk( )结束。
- 如果缓冲区中没有包含最新数据，就调用 ll_rw_block( )函数启动读操作。
- 等待，直到数据传送完成为止。这是通过调用一个名为 wait_on_buffer( )的函数来实现的，该函数把当前进程插入 b_wait 等待队列中，并挂起当前进程直到这个缓冲区被开锁

为止。

breada( )和 bread( )十分类似，但是它除了读取所请求的块之外，还要另外预读一些其他块。注意不存在把块直接写入磁盘的函数。写操作永远都不会成为系统性能的瓶颈，因为写操作通常都会延时。

2．ll_rw_block( )函数

ll_rw_block( )函数产生块设备请求；内核和设备驱动程序的很多地方都会调用这个函数。该函数的原型如下：

```
void  ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
```

其参数的含义如下。

- 操作类型 rw，其值可以是 READ、WRITE、READA 或者 WRITEA。最后两种操作类型和前两种操作类型之间的区别在于，当没有可用的请求描述符时后两个函数不会阻塞。
- 要传送的块数 nr。
- 一个 bhs 数组，有 nr 个指针，指向说明块的缓冲区首部（这些块的大小必须相同，而且必须处于同一个块设备）。

该函数的代码在 block/ll_rw_blk.c 中：

```
void ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
{
        unsigned int major;
        int correct_size;
        int i;

        if (!nr)
                return;

        major = MAJOR(bhs[0]->b_dev);

        /* Determine correct block size for this device. */
        correct_size = get_hardsect_size(bhs[0]->b_dev);

        /* Verify requested block sizes. */
        for (i = 0; i < nr; i++) {
                struct buffer_head *bh = bhs[i];
                if (bh->b_size % correct_size) {
                        printk(KERN_NOTICE "ll_rw_block: device %s: "
                                "only %d-char blocks implemented (%u)\n",
                                kdevname(bhs[0]->b_dev),
                                correct_size, bh->b_size);
                        goto sorry;
                }
        }

        if ((rw & WRITE) && is_read_only(bhs[0]->b_dev)) {
                printk(KERN_NOTICE "Can't write to read-only device %s\n",
                        kdevname(bhs[0]->b_dev));
                goto sorry;
        }
```

```
        for (i = 0; i < nr; i++) {
                struct buffer_head *bh = bhs[i];

                /* Only one thread can actually submit the I/O. */
                if (test_and_set_bit(BH_Lock, &bh->b_state))
                        continue;

                /* We have the buffer lock */
                atomic_inc(&bh->b_count);
                bh->b_end_io = end_buffer_io_sync;

                switch(rw) {
                case WRITE:
                        if (!atomic_set_buffer_clean(bh))
                                /* Hmmph! Nothing to write */
                                goto end_io;
                        __mark_buffer_clean(bh);
                        break;

                case READA:
                case READ:
                        if (buffer_uptodate(bh))
                                /* Hmmph! Already have it */
                                goto end_io;
                        break;
                default:
                        BUG();
        end_io:
                        bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
                        continue;
                }

                submit_bh(rw, bh);
        }
        return;

sorry:
        /* Make sure we don't get infinite dirty retries.. */
        for (i = 0; i < nr; i++)
                mark_buffer_clean(bhs[i]);
}
```

下面对该函数给予解释。

进入 ll_rw_block()以后，先对块大小作一些检查；如果是写访问，则还要检查目标设备是否可写。内核中有个二维数组 ro_bits，定义于 drivers/block/ll_rw_blk.c 中：

```
static long ro_bits[MAX_BLKDEV][8];
```

每个设备在这个数组中都有个标志，通过系统调用 ioctl()可以将一个标志位设置成 1 或 0，表示相应设备为只读或可写，而 is_read_only（）就是检查这个数组中的标志位是否为 1。

接下来，就通过第 2 个 for 循环依次处理对各个缓冲区的读写请求了。对于要读写的每个块，首先将其缓冲区加上锁，还要将其 buffer_head 结构中的函数指针 b_end_io 设置成指向 end_buffer_io_sync，当完成对给定块的读写时，就调用该函数。此外，对于待写的缓冲区，其 BH_Dirty 标志位应该为 1，否则就不需要写了，而既然写了，就要把它清 0，并通过 __mark_buffer_clean(bh)将缓冲区转移到干净页面的 LRU 队列中。反之，对于待读的缓冲区，其 buffer_uptodate () 标志位为 0，否则就不需要读了。每个具体的设备就好像是个服务器，所以最后具体的读写是通过 submit_bh () 将读写请求提交各“服务器”完成的，每次读写一个块，该函数的代码也在同一文件中，读者可以自己去读。

11.3.4　RAM 盘驱动程序的实现

1．RAM 盘的硬件

利用 RAM 盘的驱动程序可以访问内存的任何部分，它的主要用途是保留一部分内存并象普通磁盘一样来使用它。

RAM 盘的思想很简单，块设备是有两个操作的命令的存储介质：即写数据块和读数据块。通常这些数据存储于旋转存储设备上如软盘和硬盘，RAM 盘则简单得多，它利用预先分配的主存来存储数据块。因此不存在像磁盘那样的寻道操作，其读写操作只是在内存间进行的。RAM 盘具有快速存取的优点（没有寻道和旋转延迟的时间），适合于存储需要频繁存取的数据。

操作系统根据对 RAM 盘的需求为它分配内存的大小，RAM 盘被分成几块，每块的大小同实际磁盘的块的大小相同。一个 RAM 盘驱动程序支持将存储器中的若干区域当作 RAM 盘来使用，不同的 RAM 盘用从设备号来区分。

2．Linux 中 RAM 盘的驱动程序

RAM 盘的驱动程序同其他所有的驱动程序一样都是由一组函数组成，对 RAM 盘的操作实际上是对内存的操作，它不需要中断机制，故 RAM 盘的驱动程序不包括中断服务子程序.。一般我们对于一个驱动程序的分析是在了解硬件的基础上从该设备所提供的操作入手的，相应的写驱动程序也应该是这样的。

下面是 RAM 盘操作的结构：

```
s static struct block_device_operations rd_bd_op = {
        owner:          THIS_MODULE,
        open:           rd_open,
        ioctl:          rd_ioctl,
 };
```

在 Linux 中，RAM 盘的主设备号是 1。在 rd_open ( ) 函数中，它首先检测设备号 INITRD_MINOR，由于 INITRD 是在系统一启动的时候就已经创建，其中映像的是操作系统从偏移地址 0 开始的内容，即内核空间，如果是内核空间，其接口需要相应的发生变换即：filp->f_op = &initrd_fops。

```
static struct file_operations initrd_fops = {
```

```
        read:           initrd_read,
        release:        initrd_release,
 };
```

对于 INITRD 盘的操作用户只有读和释放的权限而无写的权限。initrd_read（）函数执行的是从内核区进行的读操作，故而是利用 memcpy_tofs（buf，（char *）initrd_start+file->f_pos，count）去完成的。initrd_release（）函数在判断没有用户操作这个设备之后，以页的方式把 INITRD 盘所占的内存释放掉。

在普通 RAM 盘接口中的另一个函数为 rd_ioctl（），同其他设备驱动程序一样是执行一些输入/输出的控制操作。

RAM 盘的驱动程序可以以模块的形式进行编译，所以驱动程序中还有一些关于模块的操作，关于模块的知识请参见上一章。

int init_module(void)；   /*执行 rd_init()*/

void cleanup_module(void) 释放模块的时候首先要把保护的缓冲区标志为无效，然后取消 ramdisk 的注册。

RAM 盘中还有 3 个函数比较重要，如下所述。

（1） int identify_ramdisk_image(kdev_t device, struct file *fp, int start_block)；

检测设备中被映像文件的文件系统的类型，返回被映像的最大块数。

（2）static void rd_load_image(kdev_t device,int offset)

把文件映像到 RAM 盘，从偏移地址 offset 开始。

（3）void rd_load（）

用软盘启动的时候装载映像文件到 ROOT_DEV 中。

至此，我们对于 Linux 中关于 RAM 的实现有一个大体的了解，下面我们再看一个较复杂的驱动程序——硬盘的驱动程序。

11.3.5 硬盘驱动程序的实现

1．磁盘硬件

所有实际的磁盘都组织成许多柱面，每个柱面上的磁道数和磁头数相同。磁道又被划分成许多扇区。如果每条磁道上的扇区数相同，则外圈磁道的数据的密度就会小一些，这就意味着会牺牲一些磁盘容量，也意味着必须存在更复杂的系统。现代大容量的硬盘中外圈磁道有的扇区数比内圈多，这就是 IDE（Integrated Drive Electronics）驱动器，它内置的电子器件屏蔽了复杂的细节，对于操作系统来说仍呈现出简单的几何结构，每条磁道具有相同的扇区。

下面我们看一下硬盘控制器的硬件结构，以便于我们进一步了解硬盘驱动程序。对于驱动程序，了解硬盘的各种寄存器以及寄存器的各个位是重要的，表 11.1(a)～(e)给出各个寄存器的具体描述。

表 11.1　　　　　　　　(a) IDE 硬盘的控制寄存

A2	A1	A0	读功能	写功能
0	0	0	数据	数据
0	0	0	出错	写预补偿
0	0	0	扇区计数	扇区计数
0	1	1	扇区号（0～7）	扇区号（0～7）
1	0	0	柱面号低位（8～15）	柱面号低位（8～15）
1	0	0	柱面号高位（16～23）	柱面号高位（16～23）
1	1	0	选择驱动器磁头（24～27）	选择驱动器磁头（24～27）
1	1	1	状态	状态

表 11.1　(b)驱动器/磁头寄存器的选择域

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA：0 = 柱面数/磁头/扇区模式
　　1 = 逻辑寻块模式
D：　0 = 主驱动器
　　1 = 从驱动器
HSn：CHS 模式：在 CHS 模式下磁头选择
　　LBA 模式：块选择位 24～27

表 11.1　(c)出错标志寄存器各个域的功能

位	功能
D0	当它置 1 时，表明找到了扇区但不能找到数据地址的标记
D1	当它置 1 时，表明回零道命令时，发生了 1024 个脉冲仍未找到 0 道
D2	当它置 1 时，表明无效命令
D4	当它置 1 时，表示磁盘转了 8 圈仍未找到所要求的参数或出现 ID 段 CRC 错
D6	当它置 1 时，表明数据段 CRC 错或未找到数据地址标志
D7	当它置 1 时，表明在 ID 段有坏块标志

表 11.1　(d)状态寄存器各个域的含义

位	功能
D0	当它置位时，,标志错误寄存器存在错误标志
D1	当它置位时，表示正在进行命令不能接受新的命令
D2	未用
D3	当它置位时，表明 WDC 请求与主机交换数据
D4	寻找完成标志
D5	当它置位时，表示有写错误
D6	当它置位时，表示盘准备好
D7	命令写入时将它置位，命令结束时将它清除

表 11.1　　(e)命令寄存器的功能

命令	D7D6D5D4D3D2D1D0
回零道	0001R3R2R1R0
寻道	1111R3R2R1R0
读扇区	0010IMOT
写扇区	00110MOT
扫描 ID	01000000
格式化	01010000

R3～R0 的编码规定了步进脉冲的间隔

I 是中断允许位当 I=0 时，将在 BDRQ（BDRQ 为扇区缓冲区数据请求输出信号）有效时产生中断

当 I=1 时，是在命令结束时产生中断

M 是多扇区读写标志 当 M=0 时，只写一个扇区

当 M=1 时，传送多个扇区

T 是允许重试命令 当 T=0 时，允许重试

当 T=1 时，不允许重试

上面对于硬盘控制器的几个寄存器的功能作了简要的说明，由此我们可以了解到，对于硬盘的很大一部分工作，由硬盘控制器就可以完成。而对于软盘来说，其控制器则简单得多，我们需要编程去完成各种功能，这样软盘驱动程序就变得比较复杂。在这儿我们只讨论硬盘驱动程序。

2．Linux 中硬盘驱动程序的实现

接下来我们将要讨论的驱动程序在 drivers/ide/hd.c 中，在文件为 include/linux/hdreg.h 中，定义了控制器寄存器、状态位和命令、数据结构和原形。这些宏定义可以根据其名字并结合上面所说的硬件内容去理解。

Linux 中，硬盘被认为是计算机的最基本的配置，所以在装载内核的时候，硬盘驱动程序必须就被编译进内核，不能作为模块编译。硬盘驱动程序提供内核的接口为：

```
static struct block_device_operations hd_fops = {
      open:          hd_open,
      release:       hd_release,
      ioctl:         hd_ioctl,
};
```

对硬盘的操作只有 3 个函数。我们来看一下 hd_open（ ）和 hd_release（ ）函数，打开操作首先检测了设备的有效性，接着测试了它的忙标志，最后对请求硬盘的总数加 1，来标识对硬盘的请求个数，hd_release（ ）函数则将请求的总数减 1。

前面说过，对于块设备的读写操作是先对缓冲区操作，但是当需要真正同硬盘交换数据的时候，驱动程序又干了些什么？在 hd.c 中有一个函数 hd_out（ ），可以说它在实际的数据交换中起着主要的作用。它的原形是：

```
static void  hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
                    unsigned int head,unsigned int cyl,unsigned int cmd,
```

```
void (*intr_addr)(void));
```

其中参数 drive 是进行操作的设备号；nsect 是每次读写的扇区数；sect 是读写的开始扇区号；head 是读写的磁头号；cmd 是操作命令控制命令字。

通过这个函数向硬盘控制器的寄存器中写入数据，启动硬盘进行实际的操作。同时这个函数也配合完成 cmd 命令相应的中断服务子程序，通过 SET_INIT(intr_addr) 宏定义将其地址赋给 DEVICE_INTR。

hd_request () 函数就是通过这个函数进行实际的数据交换，同其他驱动程序不同的是该函数还要根据每个命令的不同来确定一些参数，最基本的是读写方式的确定，关于硬盘的读写方式有两种，一种是单扇区的读写，另一种是多扇区的读写，单扇区的读写是指每次操作只对一个扇区操作，而多扇区则指每次对多个扇区进行操作，不同的方式其中断服务子程序不同，其相应的地址就作为参数传给 hd_out ( )，由它设置 DEVICE_INIT。hd_request( )函数确定的其他参数也就是 hd_out()所需要的参数。

我们知道块设备的实际数据交换需要中断服务子程序的配合，在本驱动程序中的中断服务子程序有以下几个主要函数。

(1) void unexpected_hd_interrupt(void)

功能：对不期望的中断进行处理(设置 SET_TIMER)。

(2) static void bad_rw_intr(void)

功能：当硬盘的读写操作出现错误时进行处理。

- 每重复 4 次磁头复位；
- 每重复 8 次控制器复位；
- 每重复 16 次放弃操作。

(3) static void recal_intr(void)

功能：重新进行硬盘的本次操作。

(4) static void read_intr(void)

功能：从硬盘读数据到缓冲区。

(5) static void write_intr(void)

功能：从缓冲区读数据到硬盘。

(6) static void hd_interupt(void)

功能：决定硬盘中断所要调用的中断程序。

在注册的时候，同硬盘中断联系的是 hd_interupt ( )，也就是说当硬盘中断到来的时候，执行的函数是 hd_interupt ( )，在此函数中调用 DEVICE_INTR 所指向的中断函数，如果 DEVICE_INTR 为空，则执行 unexpected_hd_interrupt()函数。

对硬盘的操作离不开控制寄存器，为了控制磁盘要经常去检测磁盘的运行状态，在本驱动程序中有一系列的函数是完成这项工作的，check_status ( ) 检测硬盘的运行状态，如果出现错误则进行处理。contorller_ready ( ) 检测控制器是否准备好。drive_busy ( ) 检测硬盘设备是否处于忙态。当出现错误的时候，由 dump_status ( ) 函数去检测出错的原因。wait_DRQ ( ) 对数据请求位进行测试。

当硬盘的操作出现错误的时候，硬盘驱动程序会把它尽量在接近硬件的地方解决掉，其方法是进行重复操作，这些在 bad_rw_intr ( ) 中进行，与其相关的函数有

reset_controller（ ）和 reset_hd（ ）。

函数 hd_init（ ）是对硬盘进行初始化的，这个函数的过程同其他块设备基本一致。还有一些对硬盘的参数进行测试和确定的函数，这里就不一一说明。

通过对 RAM 盘和硬盘驱动程序的分析我们对一个块设备的驱动程序应已经有一个大概的认识，那么对于其他的块设备驱动程序我们就可以根据硬盘驱动程序的分析方法去分析，在分析的时候要切记不能脱离硬件控制器，也不能一开始就扎入技术细节，那样我们就会陷入一个不可自拔的泥潭。如果要写一个块设备的驱动程序，我们除了要了解硬件寄存器外还要弄清具体驱动程序所要解决的问题，然后再根据驱动程序的写法去做进一步的工作。

11.4 字符设备驱动程序

我们说，传统的 UNIX 设备驱动是以主／从设备号为主的，每个设备都有唯一的主设备号和从设备号，而内核中则有块设备表和字符设备表，根据设备的类型和主设备号便可在设备表中找到相应的驱动函数，而从设备号则一般只用作同类型设备中具体设备项的编号。但是，由于字符设备的多样性，有时候也用从设备号作进一步的归类。这方面典型的例子就是终端设备 TTY。TTY 设备是字符设备，主设备号为 4，但是当从设备号为 0 时就表示当前虚拟控制终端，而 1～63 表示 63 个可能的虚拟控制终端，64～255 则表示 192 个可能的串口 URAT（通用异步收发器）和连接在上面的实际终端设备。这里所谓实际终端设备通常是指老式的 CRT 终端，或“笨终端”，而虚拟控制终端则通常是建立在 PC 机的显示器和图形接口卡基础上的。显然，这里相同的主设备号并不意味着相同的驱动程序。另一个例子是主设备号为 1 的“字符设备”，当从设备号为 1 时表示物理内存/dev/mem，为 2 时则表示内核的虚存空间／dev/kmem/，为 3 时表示“空设备”/dev/null，而从设备号 8 则表示随机数生成器/dev/random。类似的情况在块设备中也有（例如软盘设备），但是很少。随同 Linux 内核发布的 Documentation/devices.txt 文件中列举了对块设备和字符设备两种主设备号和从设备号的分配和指定，读者可以参阅。

11.4.1 简单字符设备驱动程序

我们来看一个最简单的字符设备，即“空设备”/dev/null。大家知道，应用程序在运行的过程中，一般都要通过其预先打开的标准输出通道或标准出错通道在终端显示屏上输出一些信息，但是有时候（特别是在批处理中）不宜在显示屏上显示信息，又不宜将这些信息重定向到一个磁盘文件中，而要求直接使这些信息流入“下水道”而消失，这时候就可以用／dev/null 来起这个“下水道”的作用，这个设备的主设备号为 1。

如前所述，主设备号为 1 的设备其实不是“设备”，而都是与内存有关，或是在内存中（不必通过外设）就可以提供的功能，所以其主设备号标识符为 MEM_MAJOR，其定义于 include/linux/major.h 中：

```
#define MEM_MAJOR        1
```

其 file_operatins 结构为 memory_fops，定义于 dreivers/char/mem.c 中：

```
static struct file_operations memory_fops = {
        open:           memory_open,    /* just a selector for the real open */
};
```

因为主设备号为 1 的字符设备并不能唯一地确定具体的设备驱动程序，因此需要根据从设备号来进行进一步的区分，所以 memory_fops 还不是最终的 file_operations 结构，还需要由 memory_open（）进一步加以确定和设置，其代码在同一文件中：

```
static int memory_open(struct inode * inode, struct file * filp)
{
         switch (MINOR(inode->i_rdev)) {
                 case 1:
                        filp->f_op = &mem_fops;
                         break;
                case 2:
                       filp->f_op = &kmem_fops;
                        break;
                 case 3:
                        filp->f_op = &null_fops;
                         break;
                       …
                }
      if (filp->f_op && filp->f_op->open)
                return filp->f_op->open(inode,filp);
            return 0;
    }
```

因为/dev/null 的从设备号为 3，所以其 file_operations 结构为 null_fops：

```
    static struct file_operations null_fops = {
         llseek:         null_lseek,
         read:           read_null,
         write:          write_null,
  };
```

由于这个结构中函数指针 open 为 NULL，因此在打开这个文件时没有任何附加操作。当通过 write()系统调用写这个文件时，相应的驱动函数为 write_null()，其代码为：

```
static ssize_t write_null(struct file * file, const char * buf,
                           size_t count, loff_t *ppos)
{
         return count;
 }
```

从中可以看出，这个函数什么也没做，仅仅返回 count，假装要求写入的字节已经写好了，而实际把写的内容丢弃了。

再来看一下读操作又做了些什么，read_null（）的代码为：

```
static ssize_t read_null(struct file * file, char * buf,
                        size_t count, loff_t *ppos)
{
         return 0;
}
```

返回 0 表示从这个文件读了 0 个字节，但是并没有到达（永远也不会到达）文件的末尾。

当然，字符设备的驱动程序不会都这么简单，但是总的框架是一样的。

11.4.2 字符设备驱动程序的注册

具有相同主设备号和类型的每类设备文件都是由 device_struct 数据结构来描述的，该结构定义于 fs/devices.c：

```
struct device_struct {
        const char * name;
        struct file_operations * fops;
};
```

其中，name 是某类设备的名字，fops 是指向文件操作表的一个指针。所有字符设备文件的 device_struct 描述符都包含在 chrdevs 表中：

```
static struct device_struct chrdevs[MAX_CHRDEV];
```

该表包含有 255 个元素，每个元素对应一个可能的主设备号，其中主设备号 255 为将来的扩展而保留的。表的第一项为空，因为没有一个设备文件的主设备号是 0。

chrdevs 表最初为空。register_chrdev( )函数用来向其中的一个表中插入一个新项，而 unregister_chrdev( )函数用来从表中删除一个项。我们来看一下 register_chrdev ()的具体实现：

```
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
 {
         if (major == 0) {
                 write_lock(&chrdevs_lock);
                 for (major = MAX_CHRDEV-1; major > 0; major--) {
                         if (chrdevs[major].fops == NULL) {
                                 chrdevs[major].name = name;
                                 chrdevs[major].fops = fops;
                                 write_unlock(&chrdevs_lock);
                                 return major;
                         }
                 }
                 write_unlock(&chrdevs_lock);
                return -EBUSY;
         }
         if (major >= MAX_CHRDEV)
                return -EINVAL;
         write_lock(&chrdevs_lock);
         if (chrdevs[major].fops && chrdevs[major].fops != fops) {
                 write_unlock(&chrdevs_lock);
                return -EBUSY;
         }
         chrdevs[major].name = name;
         chrdevs[major].fops = fops;
         write_unlock(&chrdevs_lock);
         return 0;
}
```

从代码可以看出，如果参数 major 为 0，则由系统自动分配第 1 个空闲的主设备号，并把设备名和文件操作表的指针置于 chrdevs 表的相应位置。

例如，可以按如下方式把并口打印机驱动程序的相应结构插入到 chrdevs 表中：

```
register_chrdev(6, "lp", &lp_fops);
```

该函数的第 1 个参数表示主设备号，第 2 个参数表示设备类名，最后一个参数是指向文件操作表的一个指针。

如果设备驱动程序被静态地加入内核，那么，在系统初始化期间就注册相应的设备文件类。但是，如果设备驱动程序作为模块被动态装入内核，那么，对应的设备文件在装载模块时被注册，在卸载模块时被注销。

字符设备被注册以后，它所提供的接口，即 file_operations 结构在 fs/devices.c 中定义如下：

```
/*
  * Dummy default file-operations: the only thing this does
  * is contain the open that then fills in the correct operations
  * depending on the special file...
  */
 static struct file_operations def_chr_fops = {
        open:           chrdev_open,
};
```

由于字符设备的多样性，因此，这个缺省的 file_operations 仅仅提供了打开操作，具体字符设备文件的 file_operations 由 chrdev_open（）函数决定：

```
  /*
  * Called every time a character special file is opened
  */
int chrdev_open(struct inode * inode, struct file * filp)
{
        int ret = -ENODEV;

        filp->f_op=get_chrfops(MAJOR(inode->i_rdev), MINOR(inode->i_rdev));
        if (filp->f_op) {
                ret = 0;
                if (filp->f_op->open != NULL) {
                       lock_kernel();
                        ret = filp->f_op->open(inode,filp);
                       unlock_kernel();
                }
        }
        return ret;
 }
```

首先调用 MAJOR（）和 MINOR（）宏从索引节点对象的 i_rdev 域中取得设备驱动程序的主设备号和从设备号，然后调用 get_chrfops（）函数为具体设备文件安装合适的文件操作。如果文件操作表中定义了 open 方法，就调用它。

注意，最后一次调用的 open( )方法就是对实际设备操作，这个函数的工作是设置设备。通常，open(　)函数执行如下操作。

- 如果设备驱动程序被包含在一个内核模块中，那么把引用计数器的值加 1，以便只有把设备文件关闭之后才能卸载这个模块。
- 如果设备驱动程序要处理多个同类型的设备，那么，就使用从设备号来选择合适的驱动程序，如果需要，还要使用专门的文件操作表选择驱动程序。

- 检查该设备是否真正存在，现在是否正在工作。
- 如果必要，向硬件设备发送一个初始化命令序列。
- 初始化设备驱动程序的数据结构。

11.4.3 一个字符设备驱动程序的实例

下面我们通过一个实例对字符设备以及编写驱动程序的方法进行说明，通过下面的分析我们可以了解一个设备驱动程序的编写过程以及注意事项。虽然这个驱动程序没有什么实用价值,但是我们也可以通过它对一个驱动程序的编写特别是字符设备驱动程序有一定的认识。

一个设备驱动程序在结构上是非常相似的，在 Linux 中，驱动程序一般用 C 语言编写，有时也支持一些汇编和 C++语言。

1．头文件、宏定义和全局变量

一个典型的设备驱动程序一般都包含有一个专用头文件，这个头文件中包含一些系统函数的声明、设备寄存器的地址、寄存器状态位和控制位的定义以及用于此设备驱动程序的全局变量的定义，另外大多数驱动程序还使用以下一些标准的头文件。

param.h　　包含一些内核参数
dir.h　　包含一些目录参数
user.h　　用户区域的定义
tty.h　　终端和命令列表的定义
fs.h　　其中包括 Buffer header 信息

下面是一些必要的头文件

```
#include <linux/kernel.h>
#include <linux/module.h>
#if CONFIG_MODVERSIONS==1  /* 处理 CONFIG_MODVERSIONS */
#define MODVERSIONS
#include <linux/modversions.h>
#endif
/* 下面是针对字符设备的头文件 */
#include <linux/fs.h>
#include <linux/wrapper.h>

/* 对于不同的版本我们需要做一些必要的事情*/
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,0)
#include <asm/uaccess.h>  /* for copy_to_user */
#endif

#define SUCCESS 0

/* 声明设备 */
/* 这是本设备的名字，它将会出现在 /proc/devices */
```

```
#define DEVICE_NAME "char_dev"

/* 定义此设备消息缓冲的最大长度 */
#define BUF_LEN  100

/* 为了防止不同的进程在同一个时间使用此设备，定义此静态变量跟踪其状态 */
static int Device_Open = 0；

/* 当提出请求的时候，设备将读写的内容放在下面的数组中 */
static char Message[BUF_LEN]；

/* 在进程读取这个内容的时候，这个指针是指向读取的位置*/
static char *Message_Ptr ；

/* 在这个文件中，主设备号作为全局变量以便于这个设备在注册和释放的时候使用*/
static int Major;
```

2．open（ ）函数

功能：无论一个进程何时试图去打开这个设备都会调用这个函数。

```
static int device_open(struct inode *inode,
                struct file *file)
{
  static int counter = 0;

#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif

  printk("Device: %d.%d\n",
      inode->i_rdev >> 8, inode->i_rdev & 0xFF);

/* 这个设备是一个独占设备，为了避免同时有两个进程使用这一个设备我们需要采取一定的措施*/
  if (Device_Open)
    return -EBUSY;

  Device_Open++;

/* 下面是初始化消息，注意不要使读写内容的长度超出缓冲区的长度，特别是运行在内核模式时，否则如果出现缓冲上溢则可能导致系统的崩溃*/
  sprintf(Message,
 "If I told you once, I told you %d times - %s",
   counter++,
    "Hello, world\n");

  Message_Ptr = Message;

/*当这个文件被打开的时候，我们必须确认该模块还没有被移走并且增加此模块的用户数目（在移走一个模块的时候会根据这个数字去决定可否移去，如果不是 0 则表明还有进程正在使用这个模块，不能移走）*/
  MOD_INC_USE_COUNT;
```

```
  return SUCCESS;
}
```

3. release ( ) 函数

功能：当一个进程试图关闭这个设备特殊文件的时候调用这个函数。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static int device_release(struct inode *inode,
                struct file *file)
#else
static void device_release(struct inode *inode,
                 struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

  /* 为下一个使用这个设备的进程做准备*/
  Device_Open --;

/* 减少这个模块使用者的数目，否则一旦你打开这个模块以后，你永远都不能释放掉它*/
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
  return 0;
#endif
}
```

4. read ( ) 函数

功能：当一个进程已经打开此设备文件以后并且试图去读它的时候调用这个函数。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* 把读出的数据放到这个缓冲区*/
    size_t length,   /* 缓冲区的长度*/
    loff_t *offset)  /* 文件中的偏移 */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,       int length)
#endif
{
  /* 实际上读出的字节数 */
  int bytes_read = 0;
  /* 如果读到缓冲区的末尾，则返回 0 ，类似文件的结束*/
  if (*Message_Ptr == 0)
    return 0;
  /* 将数据放入缓冲区中*/
  while (length && *Message_Ptr)  {
 /* 由于缓冲区是在用户空间而不是内核空间，所以我们必须使用 copu_to_user()函数将内核空间中
```

```
的数据拷贝到用户空间*/
        copy_to_user(buffer++,*(Message_Ptr++), length--);
        bytes_read ++;
      }

    #ifdef DEBUG
       printk ("Read %d bytes, %d left\n",
         bytes_read, length);
    #endif

       /* Read 函数返回一个真正读出的字节数*/
      return bytes_read;
    }
```

5. write ( )　函数

功能：当试图将数据写入这个设备文件的时侯，这个函数被调用。

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
static ssize_t device_write(struct file *file,
    const char *buffer,
    size_t length,
    loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)", file, buffer, length);
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
    copy_from_user(Message, buffer, length);

  Message_Ptr = Message;

  /* 返回写入的字节数 */
  return i;
}
```

6. 这个设备驱动程序提供给文件系统的接口

当一个进程试图对我们生成的设备进行操作的时候就利用下面这个结构，这个结构就是我们提供给操作系统的接口，它的指针保存在设备表中，在 init_module () 中被传递给操作系统。

```
struct file_operations Fops = {
  read:    device_read,
```

```
  write:    device_write,
  open:     device_open,
  release: device_release
  };
```

7．模块的初始化和模块的卸载

init_module 函数用来初始化这个模块——注册该字符设备。init_module （）函数调用 module_register_chrdev，把设备驱动程序添加到内核的字符设备驱动程序表中，它返回这个驱动程序所使用的主设备号。

```
int init_module()
{
  /* 试图注册设备*/
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);

  /* 失败的时候返回负值*/
  if (Major < 0) {
    printk ("%s device failed with %d\n",
        "Sorry, registering the character",
        Major);
    return Major;
  }

  printk ("%s The major device number is %d.\n",
         "Registeration is a success.",
         Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
         "and see what happens.\n");

  return 0;
}
```

以下这个函数的功能是卸载模块，主要是从 /proc 中 取消注册的设备特殊文件。

```
void cleanup_module()
{
  int ret;

  /* 取消注册的设备*/
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

  /* 如果出错则显示出错信息 */
  if (ret < 0)
    printk("Error in unregister_chrdev: %d\n", ret);
}
```

11.4.4 驱动程序的编译与装载

写完了设备驱动程序，下一项任务就是对驱动程序进行编译和装载。在 Linux 里，除了直接修改系统内核的源代码，把设备驱动程序加进内核外，还可以把设备驱动程序作为可加载的模块，由系统管理员动态地加载它，使之成为内核的一部分。也可以由系统管理员把已加载的模块动态地卸载下来。Linux 中，模块可以用 C 语言编写，用 gcc 编译成目标文件（不进行链接，作为*.o 文件存盘），为此需要在 gcc 命令行里加上-c 的参数。在编译时，还应该在 gcc 的命令行里加上这样的参数：-D__KERNEL__ -DMODULE。由于在不链接时，gcc 只允许一个输入文件，因此一个模块的所有部分都必须在一个文件里实现。

编译好的模块*.o 放在/lib/modules/xxxx/misc 下(xxxx 表示内核版本)，然后用 depmod -a 使此模块成为可加载模块。模块用 insmod 命令加载，用 rmmod 命令来卸载，并可以用 lsmod 命令来查看所有已加载的模块的状态。

编写模块程序的时候，必须提供两个函数，一个是 int init_module(void)，供 insmod 在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。init_module 返回 0 以表示初始化成功，返回负数表示失败。另一个函数是 void cleanup_module (void)，在模块被卸载时调用，负责进行设备驱动程序的清除工作。

在成功地向系统注册了设备驱动程序后(调用 register_chrdev 成功后)，就可以用 mknod 命令来把设备映射为一个特别文件，其他程序使用这个设备的时候，只要对此特别文件进行操作就行了。

第十二章　网络

Linux 的网络功能是 Linux 最显著的特点之一，它作为一种网络操作系统，具有比 Windows NT 安全稳定、简易方便的优点，在操作系统领域成为一支不可忽视的生力军。本章将以面向对象的思想为核心，分别对网络部分的 4 个主要对象：协议、套接字、套接字缓冲区及网络设备接口进行了具体分析。

在 Linux 的应用方面，基于 Linux 的网络服务器是非常成功的范例，并且广泛用于商业领域，在网络服务器平台中所占比例逐年上升，所以对 Linux 的网络部分的研究具有广阔的市场价值和现实意义。

12.1　概述

Linux 优秀的网络功能和它严密科学的设计思想是分不开的。在分析 Linux 网络内容之前，我们先大体上了解一下网络部分的设计思想及其特点，这对于我们后面的分析很有帮助。

(1) Linux 的网络部分沿用了传统的层次结构。网络数据从用户进程传输到网络设备要经过 4 个层次，如图 12.1 所示。

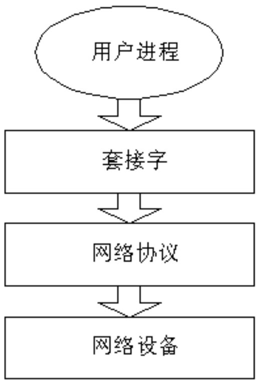


图 12.1　Linux 网络层次模型

每个层次的内部，还可以再细分为很多层次。数据的传输过程只能依照层次的划分，自顶向下进行，不能跨越其中的某个或某些层次，这就使得网络传输只能有一条而且是唯一的一条路径，这样做的目的就是为了提高整个网络的可靠性和准确性。

2. Linux 对以上网络层次的实现采用了面向对象的设计方法，层次模型中的各个层次被抽象为对象，这些对象的详细情况如下所述。

- 网络协议（protoal）。网络协议是一种网络语言，它规定了通信双方之间交换信息的一种规范，它是网络传输的基础。
- 套接字（socket）。一个套接字就是网络中的一个连接，它向用户提供了文件的 I/O，并和网络协议紧密地联系在一起，体现了网络和文件系统、进程管理之间的关系，它是网络传输的入口。
- 设备接口（device and interface）。网络设备接口控制着网络数据由软件—硬件—软件的过程，体现了网络和设备的关系，它是网络传输的桥梁。
- 套接字缓冲区（sh_buff）。网络中的缓冲器叫做套接字缓冲区。它是一块保存网络数据的内存区域，体现了网络和内存管理之间的关系，它是网络传输的灵魂。

这 4 个对象之间的关系如图 12.2 所示。

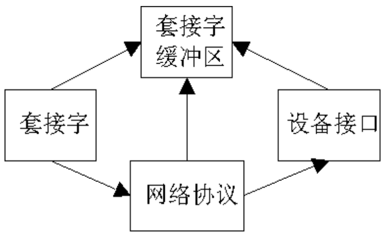


图 12.2　Linux 的网络对象及其之间的关系

从图 12.2 中我们可以看出：这 4 个对象之间的关系是非常紧密的，其中套接字缓冲区的作用非常重要，它和其他 3 个对象均有关系。本章下面的部分将对这 4 个对象及其之间的关系做详细的介绍。

Linux 网络部分为了提高它整体上的兼容性，每一个核心对象都包含了很多种类，为了便于对网络内核的分析，每一个对象我们只选择最常用的一种详细说明，其他种类从略。

12.2　网络协议

一说起网络首先就应该想到网络协议，协议是网络特有的产物，它也是整个网络传输的基础。因为网络协议非常标准规范，它在不同的系统上的用法和工作原理都是一样的，而且介绍协议的书很多，所以，协议这一节不作为本章的重点，我们只是以最常用的一种协议——TCP/IP 为例，简要介绍网络协议的工作原理和过程。

12.2.1　网络参考模型

为了实现各种网络的互连，国际标准化组织（ISO）制定了开放式系统互连（OSI）参考模型。所谓开放，就是指只要遵循标准，一个系统就可以和位于世界上任何地方遵循这一标准的其他任何系统进行通信。OSI 模型提供了一个讨论不同网络协议的参考。

尽管 OSI 的体系结构从理论上讲是比较完整的，但实际上，完全符合 OSI 各层协议的商用产品却很少进入市场。而使用 TCP/IP 的产品却大量涌入市场，几乎所有的工作站都配有

TCP/IP，使得 TCP/IP 成为计算机网络的实际的国际标准。OSI 参考模型和 TCP/IP 参考模型如表 12.1 所示。

表 12.1　　　　OSI 参考模型和 TCP/IP 参考模型

OSI 参考模型			TCP/IP 参考模型
应用层			应用层
表示层			
对话层			
传输层			传输层
网络层			Internet
数据链路层			网络接口 物理层
物理层			

12.2.2　TCP/IP 工作原理及数据流

TCP/IP 不是一个单独的协议，它是由一组协议组成的协议集，在 TCP/IP 参考模型中各层对应的协议如表 12.2 所示。

表 12.2　　　　TCP/IP 协议集

应用层	TELNET　FTP　SMTP　DNS
传输层	TCP　　UDP
Internet	IP
网络接口 物理层	ARPANET　　SATNET PPP/SLIP　　LAN

其中最主要的就是 IP（Internet 协议）和 TCP（传输控制协议）。

12.2.3　Internet 协议

IP 不仅是 TCP/IP 的一个重要组成部分，而且也是 OSI 模型的一个基本协议。

IP 定义了一个协议，而不是一个连接，因此与网络连接无关。IP 主要负责数据报在计算机之间的寻址问题，并管理这些数据报的分段过程。该协议在信息数据报格式和由数据报信息组成的报头方面有规范的定义。IP 负责数据报的路由，决定数据报发送到哪里以及在出现问题时更换路由。

IP 数据报的传输具有“不可靠性”，数据报的传输不能受到保障，因为数据报可能会遇到延迟或路由错误，或在数据报分解和重组时遭到破坏。IP 没有能力证实发送的报文是否能被正确的接收，IP 把验证和流量控制的任务交给了分层模型中的其他部件完成。

IP 是无连接的，它不管数据报沿途经过那些节点。它的这些特点都在 IP 报体现。如图

12.3 所示，数据经过 IP 层时，都会被加上 IP 的协议头，其输入／输出是从用户的角度来看的。

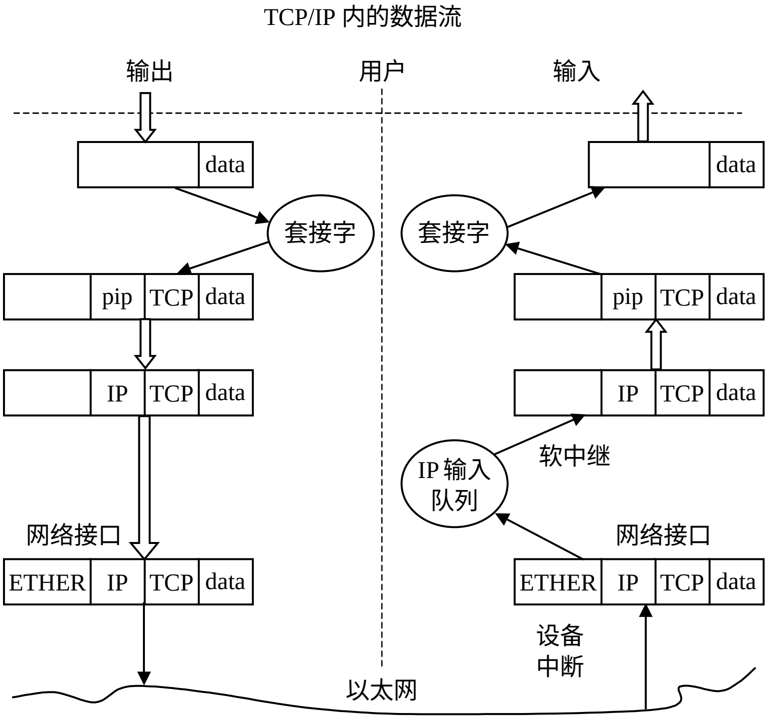


图 12.3　TCP/IP 内数据流

IP 的协议头，也可叫做 IP 数据报或 IP 报头，是 IP 的基本传输单元。IP 协议头的结构如图 12.4 所示。

<table>
<tr><td>版本号</td><td colspan="2">长度</td><td>服务类型</td><td colspan="4">数据包长度</td></tr>
<tr><td colspan="4">标识</td><td></td><td>DF</td><td>MF</td><td>标志偏移量</td></tr>
<tr><td colspan="2">生存时间</td><td colspan="2">传输</td><td colspan="4">头部校验和</td></tr>
<tr><td colspan="8">发送地址</td></tr>
<tr><td colspan="8">目标地址</td></tr>
<tr><td colspan="4">选项</td><td colspan="4">填充</td></tr>
</table>

图 12.4　IP 数据报头

12.2.4　TCP

TCP 是传输层中使用最为广泛的一协议，它可以向上层提供面向连接的协议，使上层启动应用程序，以确保网络上所发送的数据报被完整接收。就这种作用而言，TCP 的作用是提

供可靠通信的有效报文协议。一旦数据报被破坏或丢失，通常是 TCP 将其重新传输，而不是应用程序或 IP。

TCP 必须与低层的 IP（使用 IP 定义好的方法）和高层的应用程序（使用 TCP-ULP 元语）进行通信。TCP 还必须通过网络与其他 TCP 软件进行通信。为此，它使用了协议数据单元（PDU），在 TCP 用语中称为分段。

TCP PDU（通常称为 TCP 报头）的分布如图 12.5 所示。

<table>
<tr><td colspan="8">本机端口(16位)</td><td>远端端口(16位)</td></tr>
<tr><td colspan="9">序号(32位)</td></tr>
<tr><td colspan="9">确认号(32位)</td></tr>
<tr><td>数据偏移(4位)</td><td>保留(6位)</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>窗口(16位)</td></tr>
<tr><td colspan="8">校验和(16位)</td><td>紧急指针(16位)</td></tr>
<tr><td colspan="9">选项和填充</td></tr>
</table>

图 12.5　TCP 数据单元

部分域含义如下。

- 本机端口：标识本机 TCP 用户（通常为上层应用程序）的 16 位域。
- 远端端口：标识远程计算机 TCP 用户的 16 位域。
- 序号：指明当前时钟在全文中位置的序号。也可用在两个 TCP 之间以提供初始发送序号（ISS）。
- 确认号：指明下一个预计序列的序号。反过来，它还可以表示最后接收数据的序号，表示最后接收的序号加 1。
- 数据偏移：用于标识数据段的开始。
- URG：如果打开（值为 1），则指明紧急指针域有效。
- ACK：如果打开，则指明确认域有效。
- RST：如果打开，则指明要重复连接。
- SYN：如果打开，则指明要同步的序号。
- FIN：如果打开，则指明发送双方不再发送数据。这与传输结束标志是相同的。

这些域在 TCP 连接和传输数据时会用到。

TCP 对如何通信有许多规则。这些规则以及 TCP 连接、传输要遵循的过程，通常都体现在状态数据报中（因为 TCP 是一个状态驱动协议，其行为取决于状态标志或类似结构）。要完全避免复杂的状态数据报是很困难的，所以流程图对理解 TCP 是一种很有效的方法。下面我们就以 TCP 连接的流程图为例，介绍 TCP 的工作原理。如图 12.6 所示。

此过程以计算机 A 的 TCP 开始，TCP 可从它的 ULP 接收连接请求，通过它向计算机 B 发送一个主动打开原语，所构成的分段应设置 SYN 标志（值为 1），并分配一个序列号 M。图 12.6 用 SYN 50 表示，SYN 标志打开，序号 M 用 50 表示，可任意选择。

计算机 B 上的应用程序将向它的 TCP 发送一个被动打开指令，当接收到 SYN M 分段时，

计算机 B 上的 TCP 将序号 M+1 发回一个确认给计算机 A，图 12.6 用 ACK 51 表示。计算机 B 也为自己设置一个初始发送序号 N，图 12.6 用 SYN 200 表示。

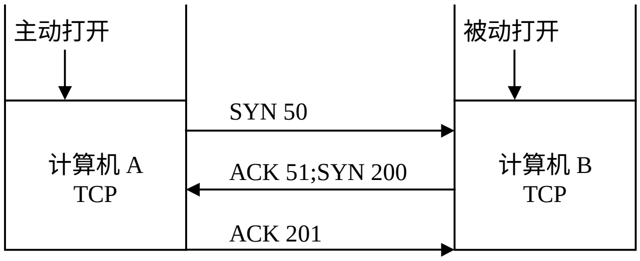


图 12.6　TCP 连接示意图

计算机 A 根据接收到的内容，通过将序号设置为 N+1 ,发回他自己的确认报文，图 12.6 用 ACK 201 表示。然后，打开并确认此次连接，计算机 A 和计算机 B 通过 ULP 将连接打开报文发送到请求的应用程序。

至此两台计算机建立了连接，可以在 TCP 层传输数据。

12.3　套接字（socket）

12.3.1　套接字在网络中的地位和作用

socket 在所有的网络操作系统中都是必不可少的，而且在所有的网络应用程序中也是必不可少的。它是网络通信中应用程序对应的进程和网络协议之间的接口，如图 12.7 所示。

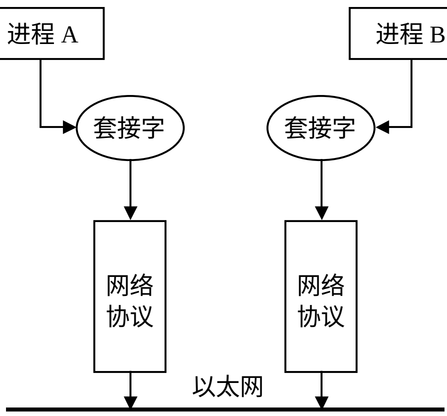


图 12.7　套接字在网络系统中的地位

socket 在网络系统中的作用如下。

（1）socket 位于协议之上，屏蔽了不同网络协议之间的差异。

（2）socket 是网络编程的入口，它提供了大量的系统调用，构成了网络程序的主体。

（3）在 Linux 系统中，socket 属于文件系统的一部分，网络通信可以被看作是对文件的读取，使得我们对网络的控制和对文件的控制一样方便。

12.3.2　套接字接口的种类

Linux 支持多种套接字种类，不同的套接字种类称为“地址族”，这是因为每种套接字种类拥有自己的通信寻址方法。Linux 所支持的套接字地址族见表 12.3。

Linux 将上述套接字地址族抽象为统一的 BSD 套接字接口，应用程序关心的只是 BSD 套接字接口，而 BSD 套接字由各地址族专有的软件支持。一般而言，BSD 套接字可支持多种套接字类型，不同的套接字类型提供的服务不同，Linux 所支持的部分 BSD 套接字类型见表 12.4，但表 12.3 中的套接字地址族并不一定全部支持表 12.4 中的这些套接字类型。

表 12.3　　Linux 支持的套接字地址族

套接字地址族	描述
UNIX	UNIX 域套接字
INET	通过 TCP/IP 协议支持的 Internet 地址族
AX25	Amater radio X25
APPLETALK	Appletalk DDP
IPX	Novell IPX
X25	X25

表 12.4　　Linux 所支持的 BSD 套接字类型

BSD 套接字类型	描述
流（stream）	这种套接字提供了可靠的双向顺序数据流，可保证数据不会在传输过程中丢失、破坏或重复出现。流套接字通过 INET 地址族的 TCP 协议实现
数据报（datagram）	这种套接字也提供双向的数据传输，但是并不对数据的传输提供担保，也就是说，数据可能会以错误的顺序传递，甚至丢失或破坏。这种类型的套接字通过 INET 地址族的 UDP 协议实现
原始（raw）	利用这种类型的套接字，进程可以直接访问底层协议（因此称为原始）。例如，可在某个以太网设备上打开原始套接字，然后获取原始的 IP 数据传输信息
可靠发送的消息	和数据报套接字类似，但保证数据被正确传输到目的端
顺序数据包	和流套接字类似，但数据包大小是固定的
数据包（packet）	这并不是标准的 BSD 套接字类型，它是 Linux 专有的 BSD 套接字扩展，可允许进程直接在设备级访问数据包

下面我们以 INET 套接字地址族、流套接字类型为例，详细介绍套接字的工作原理和通信过程。

12.3.3 套接字的工作原理

INET 套接字就是支持 Internet 地址族的套接字，它位于 TCP 之上，BSD 套接字之下，如图 12.8 所示，这里也体现了 Linux 网络模块分层的设计思想。

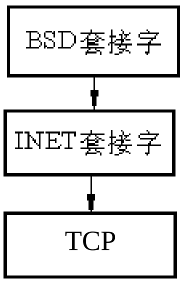


图 12.8 INET 套接字

INET 和 BSD 套接字之间的接口通过 Internet 地址族套接字操作集实现，这些操作集实际是一组协议的操作例程，在 include/linux/net.h 中定义为 proto_ops：

```
    struct proto_ops {
       int   family;

       int   (*release)      (struct socket *sock);
       int   (*bind)         (struct socket *sock, struct sockaddr *umyaddr,
                              int sockaddr_len);
       int   (*connect)      (struct socket *sock, struct sockaddr *uservaddr,
                             int sockaddr_len, int flags);
       int   (*socketpair)   (struct socket *sock1, struct socket *sock2);
       int   (*accept)       (struct socket *sock, struct socket *newsock,
                              int flags);
       int   (*getname)      (struct socket *sock, struct sockaddr *uaddr,
                              int *usockaddr_len, int peer);
    unsigned int (*poll)  (struct file *file, struct socket *sock, struct poll_table_struct
*wait);
       int   (*ioctl)        (struct socket *sock, unsigned int cmd,
                             unsigned long arg);
      int   (*listen)       (struct socket *sock, int len);
      int   (*shutdown)     (struct socket *sock, int flags);
      int   (*setsockopt)   (struct socket *sock, int level, int optname,
                              char *optval, int optlen);
      int   (*getsockopt)   (struct socket *sock, int level, int optname,
                              char *optval, int *optlen);
    int    (*sendmsg)       (struct socket *sock, struct msghdr *m, int  total_len, struct
scm_cookie *scm);
    int    (*recvmsg)      (struct socket *sock, struct msghdr *m, int total_len, int flags,
struct scm_cookie *scm);
    int    (*mmap)         (struct file *file, struct socket *sock, struct vm_area_struct * vma);
    ssize_t (*sendpage)   (struct socket *sock, struct page *page, int offset, size_t size, int
```

```
flags);
    };
```

这个操作集类似于文件系统中的 file_operations 结构。BSD 套接字层通过调用 proto_ops 结构中的相应函数执行任务。BSD 套接字层向 INET 套接字层传递 socket 数据结构来代表一个 BSD 套接字，socket 结构在 include/linux/net.h 中定义如下：

```
struct socket
{
       socket_state          state;

       unsigned long         flags;
       struct proto_ops      *ops;
       struct inode          *inode;
       struct fasync_struct  *fasync_list; /* Asynchronous wake up list    */
       struct file           *file;       /* File back pointer for gc     */
       struct sock           *sk;
       wait_queue_head_t     wait;

       short                 type;
       unsigned char         passcred;
  };
```

但在 INET 套接字层中，它利用自己的 sock 数据结构来代表该套接字，因此，这两个结构之间存在着链接关系，sock 结构定义于 include/net/sock.h（此结构有 80 多行，在此不予列出）。在 BSD 的 socket 数据结构中存在一个指向 sock 的指针 sk，而在 sock 中又有一个指向 socket 的指针，这两个指针将 BSD socket 数据结构和 sock 数据结构链接了起来。通过这种链接关系，套接字调用就可以方便地检索到 sock 数据结构。实际上，sock 数据结构可适用于不同的地址族，它也定义有自己的协议操作集 proto。在建立套接字时，sock 数据结构的协议操作集指针指向所请求的协议操作集。如果请求 TCP，则 sock 数据结构的协议操作集指针将指向 TCP 的协议操作集。

进程在利用套接字进行通信时，采用客户/服务器模型。服务器首先创建一个套接字，并将某个名称绑定到该套接字上，套接字的名称依赖于套接字的底层地址族，但通常是服务器的本地地址。套接字的名称或地址通过 sockaddr 数据结构指定，该结构定义于 include/linux/socket.h 中：

```
struct sockaddr {
        sa_family_t     sa_family;      /* address family, AF_xxx       */
        char            sa_data[14];    /* 14 bytes of protocol address */
};
```

对于 INET 套接字来说，服务器的地址由两部分组成，一个是服务器的 IP 地址，另一个是服务器的端口地址。已注册的标准端口可查看 /etc/services 文件。将地址绑定到套接字之后，服务器就可以监听请求链接该绑定地址的传入连接。连接请求由客户生成，它首先建立一个套接字，并指定服务器的目标地址以请求建立连接。传入的连接请求通过不同的协议层最终到达服务器的监听套接字。服务器接收到传入的请求后，如果能够接受该请求，服务器必须创建一个新的套接字来接受该请求并建立通信连接（用于监听的套接字不能用来建立通信连接），这时，服务器和客户就可以利用建立好的通信连接传输数据。

BSD 套接字上的详细操作与具体的底层地址族有关，底层地址族的不同实际意味着寻址

方式、采用的协议等的不同。Linux 利用 BSD 套接字层抽象了不同的套接字接口。在内核的初始化阶段，内建于内核的不同地址族分别以 BSD 套接字接口在内核中注册。然后，随着应用程序创建并使用 BSD 套接字。

内核负责在 BSD 套接字和底层的地址族之间建立联系。这种联系通过交叉链接数据结构以及地址族专有的支持例程表建立。

在内核中，地址族和协议信息保存在 inet_protos 向量中，其定义于 include/net/protocol.h：

```
        struct inet_protocol *inet_protos[MAX_INET_PROTOS];

        /* This is used to register protocols. */
    struct inet_protocol
    {
            int                    (*handler)(struct sk_buff *skb);
            void                   (*err_handler)(struct sk_buff *skb, u32 info);
            struct inet_protocol   *next;
            unsigned char          protocol;
            unsigned char          copy:1;
            void                   *data;
            const char             *name;
    };
```

每个地址族由其名称以及相应的初始化例程地址代表。在引导阶段初始化套接字接口时，内核调用每个地址族的初始化例程，这时，每个地址族注册自己的协议操作集。协议操作集实际是一个例程集合，其中每个例程执行一个特定的操作。

12.3.4 socket 的通信过程

请先看如图 12.9 所示的 socket 通信过程。

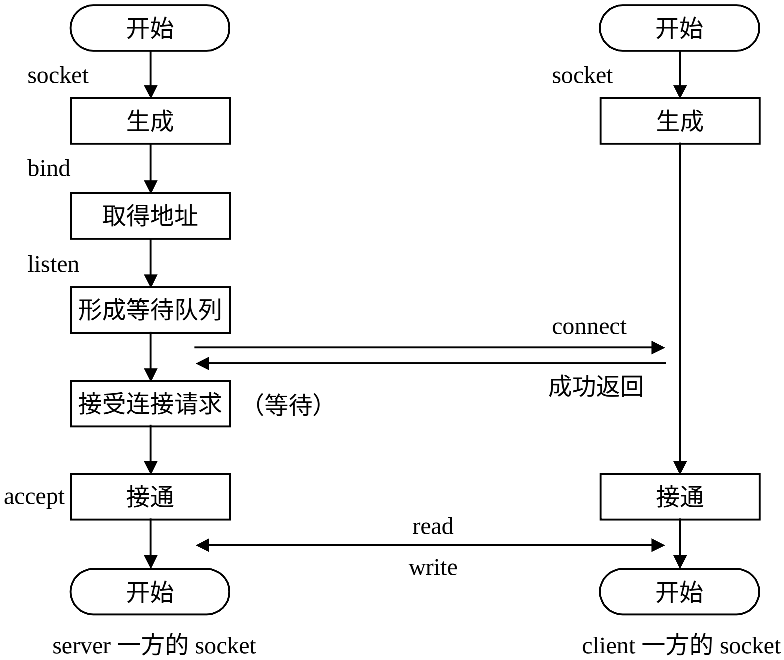


图 12.9 socket 的通信过程

1．建立套接字

Linux 在利用 socket()系统调用建立新的套接字时，需要传递套接字的地址族标识符、套接字类型以及协议，其函数定义于 net/socket.c 中：

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
        int retval;
        struct socket *sock;

        retval = sock_create(family, type, protocol, &sock);
        if (retval < 0)
                goto out;

        retval = sock_map_fd(sock);
        if (retval < 0)
                goto out_release;

out:
        /* It may be already another descriptor 8) Not kernel problem. */
        return retval;

out_release:
        sock_release(sock);
        return retval;
}
```

实际上，套接字对于用户程序而言就是特殊的已打开的文件。内核中为套接字定义了一

种特殊的文件类型，形成一种特殊的文件系统 sockfs，其定义于 net/socket.c：

```
static struct vfsmount *sock_mnt;
static DECLARE_FSTYPE(sock_fs_type, "sockfs",sockfs_read_super, FS_NOMOUNT);
```

在系统初始化时，要通过 kern_mount()安装这个文件系统。安装时有个作为连接件的 vfsmount 数据结构，这个结构的地址就保存在一个全局的指针 sock_mnt 中。所谓创建一个套接字，就是在 sockfs 文件系统中创建一个特殊文件，或者说一个节点，并建立起为实现套接字功能所需的一整套数据结构。所以，函数 sock_create()首先是建立一个 socket 数据结构，然后将其“映射”到一个已打开的文件中，进行 socket 结构和 sock 结构的分配和初始化。

新创建的 BSD socket 数据结构包含有指向地址族专有的套接字例程的指针，这一指针实际就是 proto_ops 数据结构的地址。

BSD 套接字的套接字类型设置为所请求的 SOCK_STREAM 或 SOCK_DGRAM 等。然后，内核利用 proto_ops 数据结构中的信息调用地址族专有的创建例程。

之后，内核从当前进程的 fd 向量中分配空闲的文件描述符，该描述符指向的 file 数据结构被初始化。初始化过程包括将文件操作集指针指向由 BSD 套接字接口支持的 BSD 文件操作集。所有随后的套接字（文件）操作都将定向到该套接字接口，而套接字接口则会进一步调用地址族的操作例程，从而将操作传递到底层地址族，如图 12.10 所示。

实际上，socket 结构与 sock 结构是同一事物的两个方面。如果说 socket 结构是面向进程和系统调用界面的，那么 sock 结构就是面向底层驱动程序的。可是，为什么不把这两个数据结构合并成一个呢？

我们说套接字是一种特殊的文件系统，因此，inode 结构内部的 union 的一个成分就用作 socket 结构，其定义如下：

```
struct inode  {
    …
  union {
     …
        struct socket             socket_i;
      }
  }
```

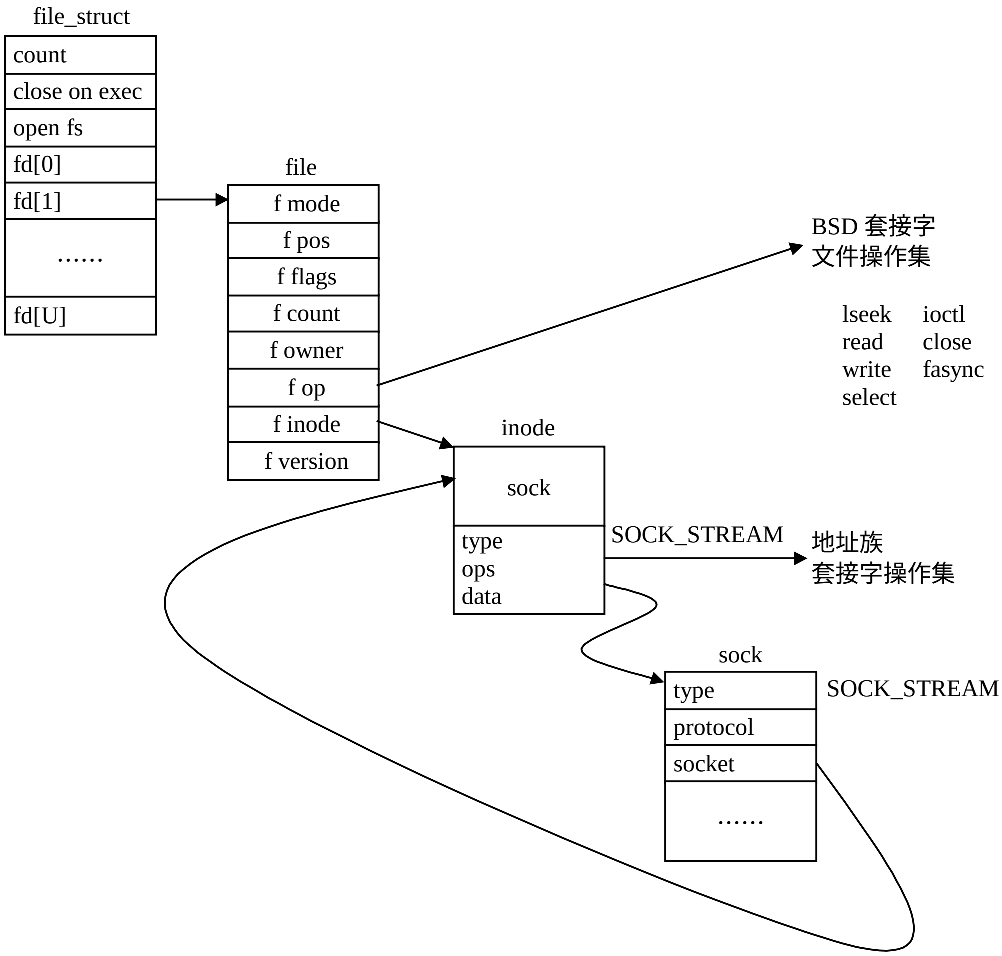


图 12.10　socket 在文件系统 inode 中的位置

由于套接字操作的特殊性，这个结构中需要大量的结构成分。可是，如果把这些结构成分全都放在 socket 结构中，则 inode 结构中的这个 union 就会变得很大，从而 inode 结构也会变得很大，而对于其他文件系统，这个 union 成分并不需要那么庞大。因此，就把套接字所需的这些结构成分拆成两部分，把与文件系统关系比较密切的那一部分放在 socket 结构中，把与通信关系比较密切的那一部分则单独组成一个数据结构，即 sock 结构。由于这两部分数据在逻辑上本来就是一体的，所以要通过指针互相指向对方，形成一对一的关系。

2．在 INET BSD 套接字上绑定（bind）地址

为了监听传入的 Internet 连接请求，每个服务器都需要建立一个 INET BSD 套接字，并且将自己的地址绑定到该套接字。绑定操作主要在 INET 套接字层中进行，还需要底层 TCP 层和 IP 层的某些支持。将地址绑定到某个套接字上之后，该套接字就不能用来进行任何其他的通信，因此，该 socket 数据结构的状态必须为 TCP_CLOSE。传递到绑定操作的 sockaddr 数据结构中包含要绑定的 IP 地址，以及一个可选的端口地址。通常而言，要绑定的地址应该是赋予某个网络设备的 IP 地址，而该网络设备应该支持 INET 地址族，并且该设备是可

用的。利用 ifconfig 命令可查看当前活动的网络接口。被绑定的 IP 地址保存在 sock 数据结构的 rcv_saddr 和 saddr 域中，这两个域分别用于哈希查找和发送用的 IP 地址。端口地址是可选的，如果没有指定，底层的支持网络会选择一个空闲的端口。

当底层网络设备接收到数据包时，它必须将数据包传递到正确的 INET 和 BSD 套接字以便进行处理，因此，TCP 维护多个哈希表，用来查找传入 IP 消息的地址，并将它们定向到正确的 socket/sock 对。TCP 并不在绑定过程中将绑定的 sock 数据结构添加到哈希表中，在这一过程中，它仅仅判断所请求的端口号当前是否正在使用。在监听操作中，该 sock 结构才被添加到 TCP 的哈希表中。

3．在 INET BSD 套接字上建立连接（connect）

创建一个套接字之后，该套接字不仅可以用于监听入站的连接请求，也可以用于建立出站的连接请求。不论怎样都涉及到一个重要的过程：建立两个应用程序之间的虚拟电路。出站连接只能建立在处于正确状态的 INET BSD 套接字上，因此，不能建立于已建立连接的套接字，也不能建立于用于监听入站连接的套接字。也就是说，该 BSD socket 数据结构的状态必须为 SS_UNCONNECTED。

在建立连接过程中，双方 TCP 要进行 3 次“握手”，具体过程在本章 12.2 节——网络协议一中有详细介绍。如果 TCP sock 正在等待传入消息，则该 sock 结构添加到 tcp_listening_hash 表中，这样，传入的 TCP 消息就可以定向到该 sock 数据结构。

4．监听（listen）INET BSD 套接字

当某个套接字被绑定了地址之后，该套接字就可以用来监听专属于该绑定地址的传入连接。网络应用程序也可以在未绑定地址之前监听套接字，这时，INET 套接字层将利用空闲的端口编号并自动绑定到该套接字。套接字的监听函数将 socket 的状态改变为 TCP_LISTEN。

当接收到某个传入的 TCP 连接请求时，TCP 建立一个新的 sock 数据结构来描述该连接。当该连接最终被接受时，新的 sock 数据结构将变成该 TCP 连接的内核 bottom_half 部分，这时，它要克隆包含连接请求的传入 sk_buff 中的信息，并在监听 sock 数据结构的 receive_queue 队列中将克隆的信息排队。克隆的 sk_buff 中包含有指向新 sock 数据结构的指针。

5．接受连接请求（accept）

接受操作在监听套接字上进行，从监听 socket 中克隆一个新的 socket 数据结构。其过程如下：接受操作首先传递到支持协议层，即 INET 中，以便接受任何传入的连接请求。相反，接受操作进一步传递到实际的协议，例如 TCP 上。接受操作可以是阻塞的，也可以是非阻塞的。接受操作为非阻塞的情况下，如果没有可接受的传入连接，则接受操作将失败，而新建立的 socket 数据结构被抛弃。接受操作为阻塞的情况下，执行阻塞操作的网络应用程序将添加到等待队列中，并保持挂起直到接收到一个 TCP 连接请求为至。当连接请求到达之后，包含连接请求的 sk_buff 被丢弃，而由 TCP 建立的新 sock 数据结构返回到 INET 套接字层，在这里，sock 数据结构和先前建立的新 socket 数据结构建立链接。而新 socket 的文件描述符（fd）被返回到网络应用程序，此后，应用程序就可以利用该文件描述符在新建

立的 INET BSD 套接字上进行套接字操作。

12.3.5 socket 为用户提供的系统调用

socket 系统调用是 socket 最有价值的一部分，也是用户唯一能够接触到的一部分，它是我们进行网络编程的接口。如表 12.5 所示。

表 12.5　　socket 系统调用

系 统 调 用	说　明
Accept	接收套接字上连接请求
Bind	在套接字绑定地址信息
Connet	连接两个套接字
Getpeername	获取已连接端套接字的地址
Getsockname	获取套接字的地址
Getsockopt	获取套接字上的设置选项
Listen	监听套接字连接
Recv	从已连接套接字上接收消息
Recvfrom	从套接字上接收消息
Send	向已连接的套接字发送消息
Sendto	向套接字发送消息
Setdomainname	设置系统的域名
Sethostid	设置唯一的主机标识符
Sethostname	设置系统的主机名称
Setsockopt	修改套接字选项
Shutdown	关闭套接字
Socket	建立套接字通信的端点
Socketcall	套接字调用多路复用转换器
Socketpair	建立两个连接套接字

12.4 套接字缓冲区（sk_buff）

套接字缓冲区是网络部分一个重要的数据结构，它描述了内存中的一块数据区域，该数据区域存放着网络传输的数据包。在整个网络传输中，套接字缓冲区作为数据的载体，保证了数据传输的可靠和稳定，而且，网络部分各层次都和该数据结构密切相关，由此可见，套

接字缓冲区在网络传输过程中的作用举足轻重，对它的理解，将是我们对网络内核进行分析的主要内容。

12.4.1　套接字缓冲区的特点

套接字缓冲区和其他部分的缓冲区相比，它有自己的特点。在网络传输的源主机上，它创建于套接字层（其名字的来历），沿网络层自上而下传递，它先在协议层流动，最后在物理层消失，同时把它所带的数据传递给目标主机的物理层的套接字缓冲区，该缓冲区自下而上传递到目标主机的套接字层，并把数据传递给用户进程，目标主机的套接字缓冲区也同时消失。请参看图 12.11 所示的示例。

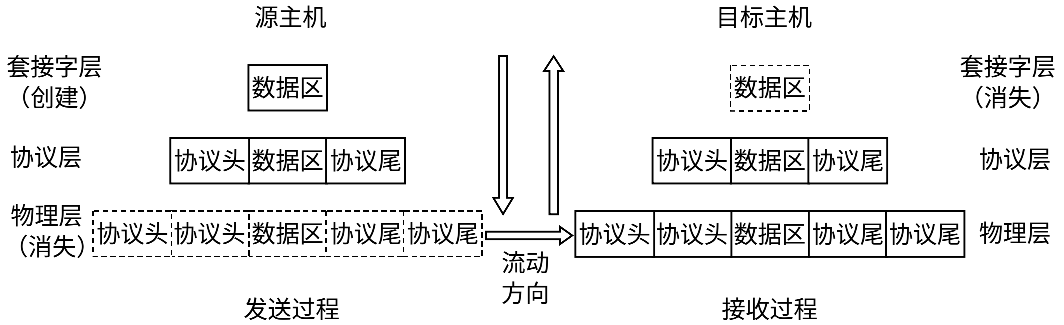


图 12.11　套接字缓冲区流程图

当套接字缓冲区在协议层流动过程中，每个协议都需要对数据区的内容进行修改，也就是每个协议都需要在发送数据时向缓冲区添加自己的协议头和协议尾，而在接收数据时去掉这些协议头和协议尾，这样就存在一个问题，当缓冲区在不同的协议之间传递时，每层协议都要寻找自己特定的协议头和协议尾，从而导致数据缓冲区的传递非常困难。我们设置 sk_buff 数据结构的主要目的就是为网络部分提供一种统一有效的缓冲区操作方法，从而可让协议层以标准的函数或方法对缓冲区数据进行处理，这是 Linux 系统网络高效运行的关键。

12.4.2　套接字缓冲区操作基本原理

在传输过程中，存在着多个套接字缓冲区，这些缓冲区组成一个链表，每个链表都有一个链表头 sk_buff_head ，链表中每个节点分别对应内存中一块数据区。因此对它的操作有两种基本方式：第 1 种是对缓冲区链表进行操作；第 2 种是对缓冲区对应的数据区进行控制。

当我们向物理接口发送数据时或当我们从物理接口接收数据时，我们就利用链表操作；当我们要对数据区的内容进行处理时，我们就利用内存操作例程。这种操作机制对网络传输是非常有效的。

前面我们讲过，每个协议都要在发送数据时向缓冲区添加自己的协议头和协议尾，而在

接收数据时去掉协议头和协议尾，那么具体的操作是怎样进行的呢？我们先看看对缓冲区操作的两个基本的函数：

```
void append_frame(char *buf, int len){
    struct sk_buff *skb=alloc_skb(len, GFP_ATOMIC);     /*创建一个缓冲区*/
   if(skb==NULL)
              my_dropped++;
   else    {
                 kb_put(skb,len);
                 memcpy(skb->data,data,len);      /*向缓冲区添加数据*/
                skb_append(&my_list, skb);  /*将该缓冲区加入缓冲区队列*/
              }
  }
void process_frame(void){
    struct sk_buff *skb;
    while((skb=skb_dequeue(&my_list))!=NULL)
     {
process_data(skb);                                        /*将缓冲区的数据传递给协
议层*/
kfree_skb(skb, FREE_READ);                            /*释放缓冲区，缓冲区从此消失*/
     }
 }
```

这两个非常简单的程序片段，虽然它们不是源程序，但是它们恰当地描述了处理数据包的工作原理，append_frame( )描述了分配缓冲区。创建数据包过程 process_frame( )描述了传递数据包，释放缓冲区的的过程。关于它们的源程序，可以去参见 net/core/dev.c 中 netif_rx( )函数和 net_bh( )函数。你可以看出它们和上面我们提到的两个函数非常相似。这两个函数非常复杂,因为他们必须保证数据能够被正确的协议接收并且要负责流程的控制,但是他们最基本的操作是相同的。

让我们再看看上面提到的函数——append_frame( )。当 alloc_skb( ) 函数获得一个长度为 len 字节的缓冲区（如图 12.12 (a)所示）后，该缓冲区包含以下内容：

- 缓冲区的头部有零字节的头部空间；
- 零字节的数据空间；
- 缓冲区的尾部有零字节的尾部空间。

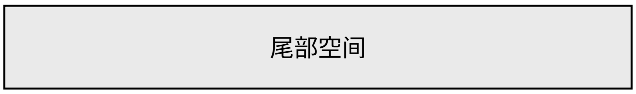
再看 skb_put( )函数（如图 12.12 (d)所示），它的作用是从数据区的尾部向缓冲区尾部不断扩大数据区大小，为后面的 memcpy( )函数分配空间。

当一个缓冲区创建以后，所有的可用空间都在缓冲区的尾部。在没有向其中添加数据之前，首先被执行的函数调用是 skb_reserve( )（如图 12.12 (b)所示），它使你在缓冲区头部指定一定的空闲空间，因此许多发送数据的例程都是这样开头的：

```
skb=alloc_skb(len+headspace, GFP_KERNEL);

skb_reserve(skb, headspace);
skb_put(skb,len);
memcpy_fromfs(skb->data,data,len);
pass_to_m_protocol(skb);
```

图 12.12 向我们展示了以上过程进行时，sk_buff 的变化情况。



(a) alloc_skb 执行后的情况



(b) alloc_reserv 执行后的情况



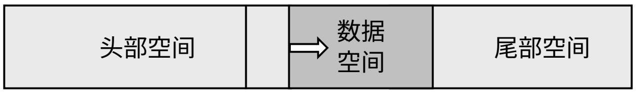
(c) sk_buff 获得数据后的情况



(d) skb_put 执行后的情况



(e) skb_push 执行后的情况



(f) skb_pull 执行后的情况

图 12.12　sk_buff 的变化过程

12.4.3　sk_buff 数据结构的核心内容

sk_buff 数据结构中包含了一些指针和长度信息，从而可让协议层以标准的函数或方法对应用程序的数据进行处理，其定义于 include/linux/skbuff.h 中：

```
struct sk_buff {
         /* These two members must be first. */
        struct sk_buff  * next;      /* Next buffer in list*/
        struct sk_buff  * prev;       /* Previous buffer in list*/

         struct sk_buff_head * list;      /* List we are on */
          struct sock     *sk;           /* Socket we are owned by */
              struct timeval  stamp;                  /* Time we arrived */
struct net_device       *dev;         /* Device we arrived on/are leaving by  */

         /* Transport layer header */
       union
```

```
        {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                 struct spxhdr   *spxh;
                 unsigned char   *raw;
         } h;

         /* Network layer header */
         union
         {
                struct iphdr    *iph;
                struct ipv6hdr  *ipv6h;
                struct arphdr   *arph;
                struct ipxhdr   *ipxh;
                 unsigned char   *raw;
        } nh;

        /* Link layer header */
         union
         {
                 struct ethhdr   *ethernet;
                 unsigned char   *raw;
         } mac;

        struct  dst_entry *dst;

        /*
          * This is the control buffer. It is free to use for every
          * layer. Please put your private variables there. If you
         * want to keep them across layers you have to do a skb_clone()
         * first. This is owned by whoever has the skb queued ATM.
          */
     char            cb[48];

     unsigned int    len;          /* Length of actual data*/
     unsigned int    data_len;
     unsigned int    csum;             /* Checksum   */
     unsigned char   __unused,       /* Dead field, may be reused  */
 cloned, /* head may be cloned (check refcnt to be sure). */
                 pkt_type,         /* Packet class   */
                ip_summed;        /* Driver fed us an IP checksum    */
   __u32           priority;      /* Packet queueing priority */
         atomic_t        users;     /* User count - see datagram.c,tcp.c */
         unsigned short  protocol;    /* Packet protocol from driver.  */
         unsigned short  security;   /* Security level of packet */
        unsigned int    truesize;         /* Buffer size */
        unsigned char   *head;         /* Head of buffer  */
        unsigned char   *data;                  /* Data head pointer
        unsigned char   *tail;                  /* Tail pointer
```

```
    unsigned char   *end;                 /* End pointer  */
        void      (*destructor)(struct sk_buff *);  /* Destruct function  */
   …
  }
```

该结构的示意图如图 12.13 所示。

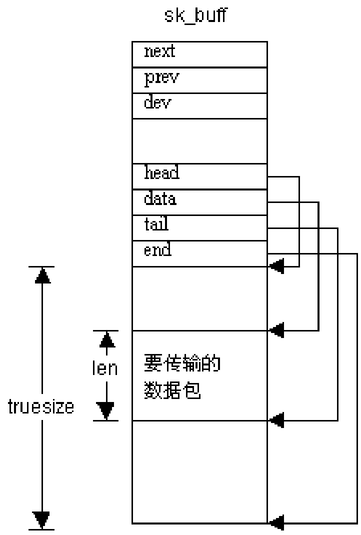


图 12.13　sk_buff 结构

每个 sk_buff 均包含一个数据块、4 个数据指针以及两个长度字段。利用 4 个数据指针，各协议层可操纵和管理套接字缓冲区的数据，这 4 个指针的用途如下所述。

head：指向内存中数据区的起始地址。sk_buff 和相关数据块在分配之后，该指针的值是固定的。

data：指向协议数据的当前起始地址。该指针的值随当前拥有 sk_buff 的协议层的变化而变化。

tail：指向协议数据的当前结尾地址。和 data 指针一样，该指针的值也随当前拥有 sk_buff 的协议层的变化而变化。

end：指向内存中数据区的结尾。和 head 指针一样，sk_buff 被分配之后，该指针的值也固定不变。

sk_buff 有两个非常重要长度字段，len 和 truesize，分别描述当前协议数据包的长度和数据缓冲区的实际长度。

12.4.4　套接字缓冲区提供的函数

1．操纵 sk_buff 链表的函数

sk_buff 链表是一个双向链表，它包括一个链表头而且每一个缓冲区都有一个 prev 和 next 指针，指向链表中前一个和后一个缓冲区节点。

```
struct sk_buff *skb_dequeue(struct skb_buff_head *list)
```

这个函数作用是把第 1 个缓冲区从链表中移走。返回取出的 sk_buff，如果队列为空，就返回空指针。添加缓冲区用到 skb_queue_head 和 skb_queue_tail 两个例程。

```
int skb_peek(struct sk_buff_head *list)
```

返回指向缓冲区链表第 1 个节点的指针。

```
int skb_queue_empty(struct sk_buff_head *list)
```

如果链表为空，返回 true 。

```
void skb_queue_head(struct sk_buff *skb)
```

这个函数在链表头部添加一个缓冲区。

```
void skb_queue_head_init(struct sk_buff_head *list)
```

初始化 sk_buff_head 结构 。该函数必须在所有的链表操作之前调用，而且它不能被重复执行。

```
__u32 skb_queue_len(struct sk_buff_head *list)
```

返回队列中排队的缓冲区的数目。

```
void skb_queue_tail(struct sk_buff *skb)
```

这个函数在链表的尾部添加一个缓冲区，这是在缓冲区操作函数中最常用的一个函数。

```
void skb_unlink(struct sk_buff *skb)
```

这个函数从链表中移去一个缓冲区。它只是将缓冲区从链表中移去，但并不释放它。

许多更复杂的协议，如 TCP 协议，当它接收到数据时，需要保持链表中数据帧的顺序或对数据帧进行重新排序。有两个函数完成这些工作：

```
void skb_append(struct sk_buff *entry, struct sk_buff *new_entry)
void skb_insert(struct sk_buff *entry, struct sk_buff *new_entry)
```

它们可以使用户把一个缓冲区放在链表中任何一个位置。

2．创建或取消一个缓冲区结构的函数

这些操作用到内存处理方法，它们的正确使用对管理内存非常重要。sk_buff 结构的数量和它们占用内存大小会对机器产生很大的影响，因为网络缓冲区的内存组合是最主要一种的系统内存组合。

```
struct sk_buff *alloc_skb(int size, int priority)
```

创建一个新的 sk_buff 结构并将它初始化。

```
void kfree_skb(struct sk_buff *skb, int rw)
```

释放一个 skb_buff。

```
struct sk_buff *skb_clone(struct sk_buff *old, int priority)
```

复制一个 sk_buff，但不复制数据部分。

```
struct sk_buff *skb_copy(struct sk_buff *skb)
```

完全复制一个 sk_buff。

3．对 sk_buff 结构数据区进行操作的操作

这些函数用到了套接字结构体中两个域：缓冲区长度(skb->len) 和缓冲区中数据包的实际起始地址 (skb->data)。这些两个域对用户来说是可见的，而且它们具有只读属性。

```
unsigned char *skb_headroom(struct sk_buff *skb)
```

返回 sk_buff 结构头部空闲空间的字节数大小。

```
unsigned char *skb_pull(struct sk_buff *skb, int len)
```

该函数将 data 指针向数据区的末尾移动，减少了 len 字段的长度。该函数可用于从接收到的数据头上移去数据或协议头。

unsigned char *skb_push(struct sk_buff *skb, int len)

该函数将 data 指针向数据区的前端移动，增加 了 len 字段的长度。在发送数据的过程中，利用该函数可在数据的前端添加数据或协议头。

unsigned char *skb_put(struct sk_buff *skb, int len)

该函数将 tail 指针向数据区的末尾移动，增加了 len 字段的长度。在发送数据的过程中，利用该函数可在数据的末端添加数据或协议尾。

unsigned char *skb_reserve(struct sk_buff *skb, int len)

该函数在缓冲区头部创建一块额外的空间，这块空间在 skb_push 添加数据时使用。因为套接字建立时并没有为 skb_push 预留空间。它也可以用于在缓冲区的头部增加一块空白区域，从而调整缓冲区的大小，使缓冲区的长度统一。这个函数只对一个空的缓冲区才能使用。

unsigned char *skb_tailroom(struct sk_buff *skb)

返回 sk_buff 尾部空闲空间的字节数大小。

unsigned char *skb_trim(struct sk_buff *skb, int len)

该函数和 put 函数的功能相反，它将 tail 指针向数据区的前端移动，减小了 len 字段的长度。该函数可用于从接收到的数据尾上移去数据或协议尾。如果缓冲区的长度比“ len ”还长，那么它就通过移去缓冲区尾部若干字节，把缓冲区的大小缩减到“ len ”长度。

12.4.5 套接字缓冲区的上层支持例程

我们上面讲了套接字缓冲区基本的操作方法，利用它们就可以完成数据包的发送和接收工作。为了保证网络传输的高效和稳定，我们需要对整个过程进行流程控制，因此，我们又引进了两个支持例程。它们是利用信号的交互来完成任务的。

sock_queue_rcv_skb（）函数用来对数据的接收进行控制，通常调用它的的形式为：

```
sk=my_find_socket(whatever);
if(sock_queue_rcv_skb(sk,skb)==-1)
{
  myproto_stats.dropped++;
  kfree_skb(skb,FREE_READ);
  return;
 }
```

它利用套接字的读队列的计数器，从而避免了大量的数据包堆积在套接字层。一旦到达这个极限，其余的数据包就会被丢弃。这样做是为了保障高层的应用协议有足够快的读取速度，比如 TCP，包含对该流程的控制，当接收端不能再接收数据时，TCP 就告诉发送端的机器停止传输。

在数据传输方面，sock_alloc_send_skb（） 可以对发送队列进行控制，我们不能把所有的缓冲区都填充数据，使得发送队列总有空余，避免了数据堵塞。这个函数在具体应用时有很多微妙之处，所以推荐编写网络协议的作者尽可能使用它。

许多发送例程利用这个函数几乎可以做所有的工作：

```
skb=sock_alloc_send_skb(sk,....)
if(skb==NULL)
```

```
        return -err;
    skb->sk=sk;
    skb_reserve(skb, headroom);
    skb_put(skb,len);
    memcpy(skb->data, data, len);
    protocol_do_something(skb);
```

上面大部分代码我们前面已经见过。其中最重要的一句是 skb->sk=sk。sock_alloc_send_skb（）负责把缓冲区送到套接字层。通过设置 skb->sk，告诉内核无论哪个例程对缓冲区进行 kfree_skb（）处理，都必须保证缓冲区已经成功地送到套接字层。因此一旦网络设备驱动程序发送一个缓冲区，并将之释放，我们就认为数据已经发送成功，这样我们就可以继续发送数据了。在源代码中我们看到 kfree_skb 操作一执行就会触发 sock_alloc_send_skb（）。

12.5　网络设备接口

Linux 网络设备接口的适用面很广，所有的 Linux 网络设备都遵循同样的接口，它提供了丰富的接口功能，但是每一个设备并不是完全都要用到这些功能。在前面我们讲过，网络部分使用的是面向对象的设计方法，每一种设备都被看成一种对象，因此在源代码中，我们就用一种具有一系列操作方法的数据结构来描述网络设备，这样做的目的就是为了在 C 语言中引入 C++的面向对象的思想。

文件 drivers/net/skeleton.c 包含了网络设备驱动程序的基本骨架。最好能找到一个最新版本的源代码放在手边，我们下边的内容将自始至终都将围绕着它展开。

12.5.1　基本结构

如图 12.14 是网络设备驱动程序的结构，从中我们可以看出，网络设备驱动程序的功能分为两部分：发送数据和接受数据。在发送数据时，设备驱动程序全权负责把来自协议层的

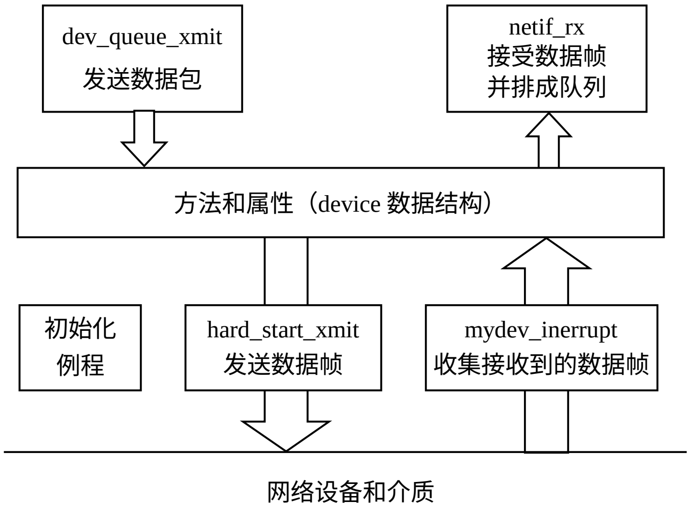


图 12.14 Linux 网络设备驱动程序的结构

网络缓冲区发送到物理介质，并且接收硬件产生的应答信号；在接收数据时，设备驱动程序接收来自网络介质上的数据帧，并把它转换成能被网络协议识别的网络缓冲区，然后把它传递给 netif_rx（ ）函数。这个函数的功能是把数据帧传递到网络协议层进行进一步的处理。

每一种网络设备驱动程序都提供了一套相应的函数，它们负责对数据的传输过程进行各种控制（包括停止、开始等），它们也负责对数据进行封装。所有这些控制信息也都保存在设备驱动程序的数据结构中。

12.5.2 命名规则

所有的 Linux 网络设备都有唯一的名字，这个名字和文件系统所规定的设备的名字没有任何联系。事实上，网络设备并没有使用文件系统的表示方法。 传统上名字只表示设备类型而不代表生产厂商，如果同一类型的网络设备有多个，它们的名字就用从 0 开始的数字加以区别，例如，如果我们装了多块以太网卡，它们的名字就是：“ eth0 ”，“ eth1 ”，“ eth2 ”等。 这种命名机制非常重要，它使得用户在编写程序和配置系统时，可以使用“ 一个以太网卡 ”来统一地表示一块网卡，而不必考虑设备来自哪个厂商 ；而且，如果我们更换网络设备，也不需重新编译内核。

下面是一些常用网络设备的命名形式：

ethn	以太网控制器，包括 10mbit/s 和 100Mbit/s
trn	令环网设备
sln	SLIP （串行接口通信协议）设备和 AX.25 KISS 方式
pppn	PPP （点对点协议）设备，包括同步和异步方式
plipn	PLIP 单元；数目和打印机端口对应。
tunln	IPIP encapsulated tunnels
nrn	NetROM 虚拟设备
isdnn	ISDN 接口 （isdn4linux）
dummyn	空设备 NULL devices
lo	回环设备

12.5.3　设备注册

每一个设备的建立都需要在设备数据结构类型中添加一个设备对象，并将它传递给 register_netdev(struct device *) 函数。这样就把你的设备数据结构和内核中的网络设备表联系起来。如果你要传递的数据结构正被内核使用，就不能释放它们，直到你卸载该设备，卸载设备用到 unregister_netdev(struct device *) 函数。这些函数调用通常在系统启动时或网络模块安装或卸载时执行。

内核不允许用同一个名字安装多个设备。因此，如果你的设备是可安装的模块，就应该利用 struct device *dev_get(const char *name) 函数来确保名字没有被使用。如果名字已经被使用，那么就必须另选一个，否则新的设备将安装失败。如果发现有设备冲突，就可以使用 unregister_netdev () 注销一个使用该名字的设备。

下面是一个典型的设备注册的源代码：

```
int register_my_device(void)
{
  int i=0;
  for(i=0;i<100;i++)
  {
    sprintf(mydevice.name,"mydev%d",i);
    if(dev_get(mydevice.name)==NULL)
    {
      if(register_netdev(&mydevice)!=0)
        return -EIO;
      return 0;
    }
  }
  printk("100 mydevs loaded. Unable to load more.<\\>n");
  return -ENFILE;
}
```

12.5.4　网络设备数据结构

网络设备数据结构 device，是网络驱动程序的最重要的部分，也是理解 Linux 网络接口的关键，它的源代码 保存在 include / linux / netdevice.h 中，这个结构比较庞大，在此不予列出，仅仅对主要的域给予解释。

所有的网络设备的信息和操作都保存在设备数据结构中。每注册一个网络设备，都需要提供数据结构中各个域的数据，这些域的含义下面将具体解释，其中，也包括对网络设备的设置。

1. 名称

name 域指网络设备的名称，我们应该按上面讨论的命名方式为设备起名。该域也可以为空，这种情况下系统自动地分配一个 ethn 名字。在 Linux 2.0 版本以后，我们可以用 dev_make_name("eth") 函数来为设备命名。

2．总线接口参数

总线接口参数用来设置设备在设备地址空间的位置。

irq：指设备使用的中断请求号（IRQ），它通常在启动时或被初始化函数时设置。如果设备没有分配中断请求号，该域可以置 0。中断请求号也可以设置为变量，由系统自动搜索一个空闲的中断请求号分配给该设备。网络设备驱动程序通常使用一个全局整型变量 irq 表示中断号，因此用户可以使用“ insmod mydevice irq=5 ”这样的命令装载一个网络设备。最后，IRQ 域也可以利用 ifconfig 命令很方便地进行设置。

base_addr（基地址）：指设备占用的基本输入输出（I/O）地址空间。如果设备没有被分配 I/O 地址或该设备运在一个没有 I/O 空间概念的系统上，该域就置 0。当该地址由用户设置时，它通常用一个全局变量 io 来表示。I/O 接口地址也可以由 ifconfig 设置。

网络设备存在着两个硬件共享内存空间的情况，例如 ISA 总线和以太网卡共享内存空间。在网络设备的 device 数据结构中有 4 个相关的域。在共享内存时，rmem_start 和 rmem_end 域就被舍弃，并且置 0；mem_start 和 mem_end 两个域标识设备共享内存块的起始地址和结束地址。如果没有共享内存的情况，上面两个域就置 0。有一些设备允许用户设置内存地址，我们通常用一个全局变量 mem 表示。

dma：标志设备正在使用的 DMA 通道。Linux 允许 DMA（像中断一样）被系统自动探测。如果没有使用 DMA 通道或 DMA 通道没有设置，该域就置 0（最新的 PC 主版上， ISA 总线 DMA 通道 0 被硬件占用，它没有和内存的刷新联系起来）。如果由用户设置 DMA 通道，通常使用一个全局变量 dma 来表示。

我们应该认识到，上面提到的这些设备硬件信息都是从用户角度来对网络设备进行控制，它们和设备的内部函数功能是一样的。如果不注册它们，它们就有可能被重用，因此设备驱动程序必须分配并注册 I/O、DMA 和中断向量这些参数。这些操作和其他设备驱动程序都用到相同的内核函数，关于具体的操作过程请参阅设备驱动程序一章相关内容。

if_port：标识一些多功能网络设备的类型，例如 combo Ethernet boards。

3．协议层参数

为了使网络协议层能智能化地执行任务，网络设备驱动程序也需要协议层提供一些性能标志和变量，这些参数都保存在设备数据结构中。

mtu：指网络接口的最大负荷，也就是网络可以传输的最大的数据包尺寸，它不包括设备自身提供的低层数据头的大小，该值常被协议层（如 IP）使用，用来选择大小合适数据包进行发送。

family：指该设备支持的地址族。常用的地址族是 AF_INET，关于地址族的概念和具体解释，请参阅本章第三节套接字。 Linux 允许一个设备同时使用多个地址族，具体情况可以参考关于 BSD 网络 API 方面的书籍。

interface hardware type ：指设备所连接的物理介质的硬件接口类型，它的值来自物理介质类型表。支持 ARP 的物理介质，它们的接口类型被 ARP 使用（参看 RCF1700）；其他的接口类型是为其他物理层定义的。新的接口类型，只有当它对内核和 net-tools（注:net-tools 也是一段源代码，它随内核一起发布，用来对 Linux 网络进行调试）都是必

需时才会添加。包含像 ifconfig 这样的工具包可以对该域进行解码。该域的定义形式为：

/*该定义来自 RFC1700（RFC 即 Request For Comments 用户数据报协议）*/

ARPHRD_NETROMARPHRD_ETHER	10mbit/s 和 100mbit/s 以太网卡
ARPHRD_EETHER	实验用网卡（没有使用）
ARPHRD_AX25	AX.25 2 级接口
ARPHRD_PRONET	PROnet token ring（没有使用）
ARPHRD_CHAOS	ChaosNET（没有使用）
ARPHRD_IEE802	802.2 networks notably token ring
ARPHRD_ARCNET	ARCnet 接口
ARPHRD_DLCI	Frame Relay DLCI
由 Linux 定义：	
ARPHRD_SLIP	Serial Line IP protocol
ARPHRD_CSLIP	SLIP with VJ header compression
ARPHRD_SLIP6	6bit encoded SLIP
ARPHRD_CSLIP6	6bit encoded header compressed SLIP
ARPHRD_ADAPT	SLIP interface in adaptive mode
ARPHRD_PPP	PPP interfaces (async and sync)
ARPHRD_TUNNEL	IPIP tunnels
ARPHRD_TUNNEL6	IPv6 over IP tunnels
ARPHRD_FRAD	Frame Relay Access Device
ARPHRD_SKIP	SKIP encryption tunnel
ARPHRD_LOOPBACK	Loopback device
ARPHRD_LOCALTLK	Localtalk apple networking device
ARPHRD_METRICOM	Metricom Radio Network

上面标注“没有使用”的接口，是因为它们虽然被定义了类型，但是目前还没有支持它们的 net-tools。Linux 内核为以太网和令环网提供了额外的支持例程。

- pa_addr：用来保持 IP 地址。
- pa_brdaddr：网络广播地址。
- pa_dstaddr：点对点连接中的目标地址。
- pa_mask：网络掩码。

上面所有域都被初始化为 0。

pa_alen：保存一个地址的长度，就 IP 地址而言，应该初始化为 4。

4．链接层变量

hard_header_len ：标识在网络缓冲区的头部，为硬件帧头准备的空间大小。这个值和将来添加的硬件帧头的字节数不一定相等。这样当 sk_buff 到达设备之前，就事先为硬件帧头准备了一块空间（scratch pad）。

在 1.2.x 系列的内核版本中，skb->data 指针指向整个缓冲区的开始，没有真正指向数据区，因此必须小心不能将“ scratch pad ”也发送出去。 这也暗示着 hard_header_len

的长度必须大于硬件帧头的长度（硬件帧头可以和数据相临）。在 1.3.x 以后的版本里，问题变得非常容易，因为 sk_buff 可以设置得足够大，不会出现空间不够的情况。至于这块空间 （scratch pad）的建立，要用到本章 12.4 节中的 skb_push( ) 函数。

物理介质的地址由分别保存在 dev_addr 和 broadcast 两个域中，我们用字符数组来保存物理地址。如果物理地址的长度比数组的长度小，那么物理地址在数组中就从左边开始保存，右边可以空余。addr_len 域用来存储物理地址的长度。因为许多介质没有物理地址，该域就置 0。还有一些其他类型的接口，物理地址必须由用户程序设置，对接口物理地址的设置可以用 ifconfig 工具。这种情况下，物理地址就不必进行初始化设置，但是我们从源代码中可以看出，如果在一个设备物理地址没有被设置，就不允许它进行传输数据。

5．接口标志

接口标志包含一些接口属性，其中一些是为了提高网络接口的兼容性而设的，因为这些标志并没有直接的用途。内核中用到的接口标志有：

- IFF_UP：接口已经激活。

在 Linux 中，IFF_RUNNIN 和 IFF_UP 标志基本上是成对出现，因为两者是相辅相成的。如果一个接口没有被标识 IFF_UP，它就不能够被删除。这和 BSD 不同，在 BSD 中，一个接口如果没有收到数据，就不会标识 IFF_UP。

- IFF_BROADCAST：设置设备广播地址有效。
- IFF_DEBUG：标识设备调试能力打开。目前并不使用。
- IFF_LOOPBACK：只有回环设备（lo）才使用该标志。
- IFF_POINTOPOINT：该设备是点对点链路设备（如 SLIP 或 PPP）。 通常点对点的连接没有子网掩码和广播地址，但是如果需要可以激活。
- IFF_RUNNING：见 IFF_UP。
- IFF_NOARP：接口不支持 ARP。这样的接口必须有一个静态的地址转换表，或者它不需要执行地址映射。NetROM 接口就是一个很好的例子。

6．数据包队列

该队列包含了等待由该网络设备发送的 sk_buff 数据包。发送到网络设备接口的数据包都由内核中协议层的代码排成队列。在每一个设备中，数据包按优先级顺序存放在 buffs [ ] 数组里。它们完全有内核代码控制，但是在启动时由设备自身进行初始化，初始化代码为：

```
int ct=0;
while(ct < DEV_NUMBUFFS)
{
    skb_queue_head_init(&dev->buffs[ct]);
    ct++;
}
```

其他域被初始化为 0。

网络设备通过设置 dev->tx_queue_len 来决定传输队列的长度。通常以太网的队列长度为 100，serial lines 为 10 。

12.5.5　支持函数

每个网络驱动程序都提供了一系列非常实用的函数，这些函数都是底层的基本的函数；每个设备还包含了一组标准的例程，协议层可以将这些例程当作设备链路层的部分而调用。关于这些函数和例程，下面我们详细介绍。

1．初始化设置（init）

init 函数在设备初始化和注册时被调用，它执行的是底层的确认和检查工作。在初始化程序里可以完成对硬件资源的配置。如果设备没有就绪或设备不能注册或其他任何原因而导致初始化工作不能正常进行，该函数就返回出错信息。一旦初始化函数返回出错信息，register_netdev（）也返回出错信息，这样该设备就不能安装。

2．打开（open）

open 这个函数在网络设备驱动程序里是网络设备被激活的时候被调用（即设备状态由down-->up）。所以实际上很多在 init 中的工作可以放到这里来做。比如资源的申请，硬件的激活。如果 dev->open 返回非零（error），则硬件的状态还是 down。 open 函数另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 open 方法里要调用 MOD_INC_USE_COUNT 宏。

3．关闭（stop）

close 函数做和 open 函数相反的工作。可以释放某些资源以减少系统负担。close 是在设备状态由 up 转为 down 时被调用的。另外如果是作为模块装入的驱动程序，close 里应调用 MOD_DEC_USE_COUNT，减少设备被引用的次数，以使驱动程序可以被卸载。另外 close 方法必须返回成功(0==success)。

4．数据帧传输例程

所有的设备驱动程序都必须提供传输例程，如果一个设备不能传输，也就没有存在的必要性。事实上，设备的所谓的传输仅仅是释放传送给它的缓冲区，而真正实现传输功能是虚拟设备。

dev->hard_start_xmit（）：该函数的功能是将网络缓冲区，也就是 sk_buff 发送到硬件设备。如果设备不能接受缓冲区，它就会返回 1，并置 dev->tbusy 为非零值。这样缓冲区就排成队列，等待着 dev->tbusy 置零以后会再次发送。如果协议层决定释放被设备抛弃的缓冲区，那么缓冲区就不会再被送回设备；如果设备知道缓冲区短时间内不被能传送，例如设备严重堵塞，那么它就调用 dev_kfree_skb（）函数丢掉缓冲区，该函数返回零值标明缓冲区已经被处理完毕。

当缓冲区被传送到硬件以后，硬件应答信号标识传输已经完毕，驱动程序必须调用dev_kfree_skb(skb，FREE_WRITE)函数释放缓冲区，一旦该调用结束，缓冲区就会很自然地消失，这样，驱动程序就不能再涉及缓冲区了。

该函数传送下来的 sk_buff 中的数据已经包含硬件需要的帧头。所以在发送方法里不需

要再填充硬件帧头，数据可以直接提交给硬件发送。sk_buff 是被锁住的（ocked）确保其他程序不会存取它。

5．硬件帧头

前面我们讲过，数据帧在传送之前先要排成对列，在加入队列之前，还要在每个数据帧的开始添加硬件帧头，这项工作对于数据传送非常必要。网络设备驱动程序提供了一个 dev->hard_header( ) 例程，来完成添加硬件帧头的工作。协议层在发送数据之前会在缓冲区的开始留下至少 dev->hard_header_len 长度字节的空闲空间。这样 dev->hard_header( ) 程序只要调用 skb_push（），然后正确填入硬件帧头就可以了。

调用这个例程需要给出和缓冲区相关的信息：设备指针、协议类型、指向源地址和目标地址（指硬件地址）的指针、数据包的长度。因为这个例程是在协议层发送函数触发之前被调用，所以一个非常重要参数值得我们注意：在这个例程中用的是 length 参数，而不是用缓冲区的长度做参数，因为调用 dev->hard_header( )时数据可能还没完全组织好。

源地址可以为“ NULL ”，这意味着“ 使用默认地址 ”；目标地址也可以为“ NULL ”，这意味着“ 目标未知 ”。如果目标地址“ 未知 ”，数据帧头的操作就不能完成，本来为硬件帧头预留的空间全部被其他信息占用,那么函数就返回填充硬件帧头空间的字节数的相反数(一定为负数）。当硬件帧头完全建立以后，函数返回所添加的数据帧头的字节数。

如果一个硬件帧头不能够完全建立，协议层必须试图解决地址问题，因为硬件地址对于数据的发送是必需的。一旦这种情况发生，dev->rebuild_header( )函数就会被调用，通常是利用 ARP（地址解析协议）来完成。如果硬件帧头还不能被解决，该函数就返回零，并且会再次尝试，协议层总是相信硬件帧头的解决是可能的。

6．数据接收

网络设备驱动程序没有关于接收的处理，当数据到来时，总是驱动程序通知系统。对一个典型的网络设备，当它收到数据后都会产生一个中断，中断处理程序调用 dev_alloc_skb（），申请一个大小合适的缓冲区（sk_buff），把从硬件传来的数据放入缓冲区。接着，设备驱动程序分析数据包的类型，把 skb->dev 设置为接收数据的设备类型 ，把 skb->protocol 设置为数据帧描述的协议类型，这样，数据帧就可以被发送到正确的协议层。硬件帧头指针保存在 skb->mac.raw 中，并且硬件帧头通过调用 skb_pull（）被去掉，因此网络协议就不涉及硬件的信息。最后还要设置 skb->pkt_type，标明链路层数据类型，设备驱动程序必须按以下类型设置 skb->pkt_type ：

PACKET_BROADCAST	链接层广播地址
PACKET_MULTICAST	链接层多路地址
PACKET_SELF	发给自己的数据帧
PACKET_OTHERHOST	发向另一个主机的数据帧（监听模式时会收到）

最后，设备驱动程序调用 netif_rx( )，把缓冲区向上传递给协议层。缓冲区首先排成一个队列，然后发出中断请求，中断请求响应后，缓冲区队列才被协议层进行处理。这种处理机制，延长了缓冲区等待处理的时间，但是减少了请求中断的次数，从而整体上提高了数据传输效率。一旦 netif_rx( )被调用，缓冲区就不在属设备驱动程序所有，它不能被修改，

而且设备驱动程序也不能再涉及它了。

在协议层，接收数据包的流程控制分两个层次：首先，netif_rx（）函数限制了从物理层到协议层的数据帧的数量。第二，每一个套接字都有一个队列，限制从协议层到套接字层的数据帧的数量。在传输方面，驱动程序的 dev->tx_queue_len 参数用来限制队列的长度。队列的长度通常是 100 帧，在进行大量数据传输的高速连接中，它足以容纳下所有等待传输的缓冲区，不会出现大量缓冲区阻塞的情况。在低速连接中，例如 slip 连接，队列的长度长设为 10 帧左右，因为传输 10 帧的数据就要花费数秒的时间排列数据。

本章的主要目的是介绍 Linux 操作系统网络部分的基本工作原理。因为 Linux 支持多种协议类型和多种网络设备，加上网络部分本身就比较复杂，所以本章所涉及的内容十分有限。为了便于说明，采用了以点带面的方法，着重介绍了网络部分的 4 个核心对象。

本章首先介绍了 Linux 网络部分源代码的面向对象设计思想，指出了 4 个核心对象：

- 网络协议；
- 套接字；
- 套接字缓冲区；
- 网络设备接口。

随后，用面向对象的分析方法，分别介绍了这 4 个核心对象的相关内容和它们之间的关系，这对于理解 Linux 网络的工作原理有很大帮助。

关于内容，本章尽可能详细具体，贴近实际应用，对于和实际应用有关系的地方都做了深入介绍。如果读者要做网络编程，请参考 12.2 网络协议，12.3 套接字，12.4 套接字缓冲区三节；如果要编写网络驱动程序，请参考 12.4 套接字缓冲区和 12.5 网络设备接口两节。

第十三章　Linux 启动系统

操作系统的启动过程既让人好奇，又让人费解。我们知道，没有操作系统的计算机是无法使用的，那么是谁把操作系统装入到内存，是操作系统自己吗？这显然是一个先有鸡还是先有蛋的问题，幸好，固化在 ROM（PC 机）中的 BIOS 帮了大忙，可以说，BIOS 是整个启动过程的先锋。实际上，尽管一台计算机的启动过程仅仅是昙花一现，但它并不简单。本章将讨论基于 i386 平台的操作系统的启动过程，其相关的保护模式的知识请参见第二章。

13.1　初始化流程

每一个操作系统都要有自己的初始化程序，Linux 也不例外。那么，怎样初始化？我们首先看一下初始化的流程。

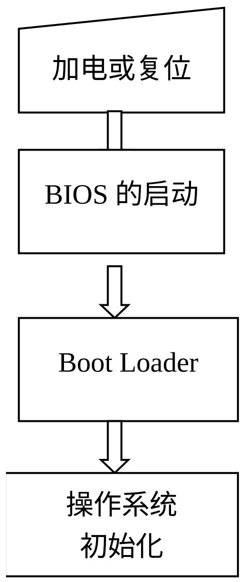


图 13.1　初始化流程

图 13.1 中的加电或复位这一项代表操作者按下电源开关或复位按钮那一瞬间计算机完成的工作。BIOS 的启动是紧跟其后的基于硬件的操作，它的主要作用就是完成硬件的初始化，稍后还要对 BIOS 进行详细的描述。BIOS 启动完成后，Boot Loader 将读操作系统代码，然后由操作系统来完成初始化剩下的所有工作。

13.1.1 系统加电或复位

当一台装有 Intel 386 CPU 的计算机系统的电源开关或复位按钮被按下时，通常所说的冷启动过程就开始了。中央处理器进入复位状态，它将内存中所有的数据清零，并对内存进行校验，如果没有错误，CS 寄存器中将置入 FFFF[0]，IP 寄存器中将置入 0000[0]，其实，这个 CS:IP 组合指向的是 BIOS 的入口，它将作为处理器运行的第一条指令。系统就是通过这个方法进入 BIOS 启动过程的。

13.1.2 BIOS 启动

BIOS 的全名是基本输入输出系统（Basic Input Output System）。它的主要任务是提供 CPU 所需的启动指令。刚才提到了，计算机进入复位状态后，内存被自动清零，CPU 此时是无法获得指令的。计算机的设计者们当然考虑到了这一点，因此，他们预先编好了供系统启动使用的启动程序，把它们存放在 ROM 中，并安排它到一个固定的位置，即 FFFF:0000，CPU 就从 BIOS 中获得了启动所需的指令集。该指令集除了完成硬件的启动过程以外，还要将软盘或硬盘上的有关启动的系统软件调入内存。

让我们看一看 BIOS 中启动程序的主要任务:首先是上电自检（POST Power-On Self Test），然后是对系统内的硬件设备进行监测和连接，并把测试所得的数据存放到 BIOS 数据区，以便操作系统在启动时或启动后使用，最后，BIOS 将从软盘或硬盘上读入 Boot Loader，到底是从软盘还是从硬盘启动要看 BIOS 的设置，如果是从硬盘启动，BIOS 将读入该盘的零柱面零磁道上的 1 扇区（MBR），这个扇区上就存放着 Boot Loader，该扇区的最后一个字存放着系统标志，如果该标志的值为 AA55，BIOS 在完成硬件监测后会把控制权交给 Boot Loader。

除了启动程序以外，BIOS 还提供一组中断以便对硬件设备的访问。我们知道，当键盘上的某一键被按下时，CPU 就会产生一个中断并把这个键的信息读入，在操作系统没有被装入以前（如 Linux 的 Bootsect.S 还没有被读入）或操作系统没有专门提供另外的中断响应程序的情况下，中断的响应程序就是由 BIOS 提供的。

这里介绍一个具体的 BIOS 系统，它的上电自检（POST）程序包含 14 个项目，具体内容如表 13.1 所示，执行过 POST 后，该系统将调入硬盘上的 Boot Loader。

表 13.1　　POST 程序包含的 14 个项目

序号	相应内容	序号	相应内容
1	CPU 处理器内部寄存器测试	8	键盘复位和测试
2	32K RAM 存储器测试	9	键盘复位和测试
3	DMA 控制器测试	10	附加 RAM 存储器测试
4	32K RAM 存储器测试	11	其他包含在系统中的 BIOS 测试
5	CRT 视频接口测试	12	软盘设备测试
6	8259 中断控制器测试	13	硬盘设备测试

7	8253 定时器测试	14	打印机接口和串行接口测试

BIOS 中的中断程序、BIOS 数据区中的信息这里就不作详细介绍了，如果你想进一步了解，请查阅相关资料，在本章后面的内容中，只会对所要涉及到的部分 BIOS 内容进行详细解释。

13.1.3　Boot Loader

Boot Loader 通常是一段汇编代码，存放在 MBR 中，它的主要作用就是将系统启动代码读入内存，有关这方面的内容相当复杂，这里请你先记住它的功能，至于详细情况，比如，怎样把系统读入，后面将会介绍。

13.1.4　操作系统的初始化

这部分实际上是初始化的关键。Boot Loader 将控制权交给操作系统的初始化代码后，操作系统所要完成的存储管理、设备管理、文件管理、进程管理等任务的初始化必须马上进行，以便进入用户态。其实不管是单任务的 DOS 操作系统还是这里介绍的多任务 Linux 操作系统，当启动过程完成以后，系统都进入用户态，等待用户的操作命令。而 Linux 要到达这个状态是相当复杂的一件工作，这一章主要就是围绕这部分内容写而。

13.2　初始化的任务

13.2.1　处理器对初始化的影响

每个操作系统都是基于计算机的硬件设备的，不管它的设计，实现，还是特性，都要依赖于一定的硬件。所有的硬件中，中央处理器（CPU）对它的影响最大。我们知道，Linux 是一个可以运行于多个不同平台的操作系统，但这并不意味着它可以抛开不同种类计算机的硬件特性。事实上，Linux 是靠在不同机器上运行不同的代码来实现跨平台特性的。Linux 巧妙地把与设备相关的代码按照设备型号分类安排，以便在编译时把对应的部分编入内核。如果你看过了在/usr/sur/arch 目录下组织的源文件，就会发现，所有 Intel 386 相关的代码在一个子目录下，而与 spark 相关的代码在另一个子目录下。代码在编译时会得到关于平台的信息，根据这个信息，编译器决定到底包含哪一段代码。

以上这些是为了说明一点，即操作系统必须支持硬件设备特别是 CPU 的特性，反过来说，硬件设备也决定操作系统的特性。

具有代表性的 Intel 80386 处理器大家一定不会陌生，它支持多任务并发执行，它的结构和机能完全是为此设计的。根据对 80386 保护模式的了解（详见第二章），我们可以看出，操作系统根据 80386 提供的机制，对计算机的资源（主存储器空间、执行时间及外围设备）进行分配和保护。通过把这些资源分配给系统中的各个任务，并对资源进行保护，是所有的

任务得以有效的运行直至完成。80386 的存储管理及保护机制，保护（系统中的）每一个任务不被另外的任务破坏。例如，操作系统通过使用存储管理机制，保证分配给不同任务的存储区互不重叠（共享存储区域除外）；通过使用保护机制，保证系统中任何一个用户任务都不能访问分配给操作系统的存储区域。

请注意，80386 提供保护机制，也提供段页式的两层内存管理，但在操作系统初始化之前，它却运行于一个既不支持保护机制，也不支持页机制的实模式下的。在这个模式下，根本没法实现多任务并发处理，所以，在一个要求实现多任务并发处理的操作系统的初始化程序中，就必须加入使 80386 进入保护模式的代码。这就是处理器对启动任务影响的一个例子。

13.2.2 其他硬件设备对处理器的影响

除了处理器以外，许多硬件设备也对初始化产生影响，刚才介绍的 BIOS 就在很大程度上影响初始化的步骤。另外，每加入一种新的硬件设备，为了它能被正常使用，你必须在操作系统中对它进行配置（PC 机的标准配置设备除外）。你或许要提到 Windows 的即插即用技术，事实上，即插即用设备是由操作系统自动完成配置的，而不是不需要配置，所以，如果你编制的是 Windows 的初始化程序，那么你需要在你的代码中加入支持自动配置即插即用设备的代码。

硬件对初始化的影响并不仅仅局限于这些方面，由于硬件在计算机系统中的中心地位，它对初始化的的影响是从始至终的。

13.3 Linux 的 Boot Loader

从这里开始，我们将对 Linux 的启动过程进程分析。首先要介绍的是 Boot Loader，因为 Boot Loader 比较复杂，并且与启动密不可分。但是要想了解 Boot Loader 的内在机制，必须从了解磁盘结构开始。

13.3.1 软盘的结构

软盘是由一个引导扇区，一个管理块（比如说 MS 的 FAT 或 EXT2 的 inode）和一个基本数据区的形式组织在一起的。由于管理块是与特定文件系统相关的，我们也可以把它归为数据区的一部分，软盘的结构如图 13.2 所示。

如果你想了解有关 inode 或数据区的相关知识，请参看第八章和第九章。在这里，我们主要关心的是引导扇区。

引导扇区中存放着用于启动的代码，以及一些有关特定文件系统的信息，在它的最后，存放着一个启动标志，如果它是 0xAA55，代表这个引导扇区是可用于启动的。

MS-DOS 的引导记录及记录内各个项目的偏移如图 13.3 所示，Linux 引导记录跟它大体一致，只是没有磁盘参数表。

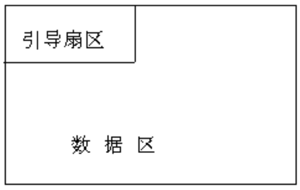


图 13.2　软盘的结构

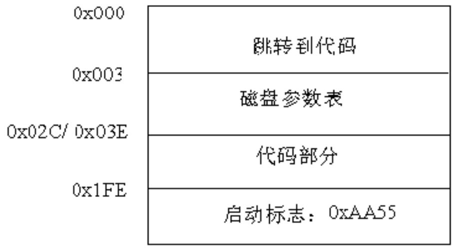


图 13.3　软盘引导区

13.3.2　硬盘的结构

硬盘相对于软盘来说要复杂一些，一个硬盘在 DOS 的文件系统下可被分为 4 个基本分区(Pramiray Partition）。如果分区数目达不到所需，那么可以把一个基本分区定义为一个扩展分区(Extended Partition)，然后再把这个基本分区分为一个或多个逻辑分区。整个硬盘有一张分区表，它存放在硬盘的第一个扇区（MBR）里，而每个扩展分区也对应一个分区表，它存放在该扩展分区对应的第一个扇区里。图 13.4 是一张硬盘分区层次图。

同样地，我们也要介绍硬盘的引导扇区内容，不过，硬盘的分区机制使它的引导扇区也有了几个层次。这里，我们先看一下整个硬盘的主引导扇区（MBR），它的结构与软盘的结构是很相似的，如图 13.5 所示。

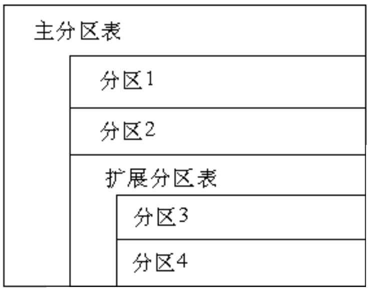


图 13.4　硬盘分区层次图

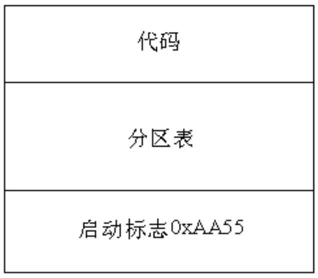


图 13.5　硬盘引导扇区

除了主引导扇区外，每个基本分区和扩展分区也有自己的引导扇区，它们的结构与主引导扇区是相同的。而逻辑分区的引导扇区通常不能用于启动。如果你使用 DOS 下的 FDISK 程序，你就会发现逻辑分区不能设置为 active 状态，也就是说 DOS 不能从该分区启动。

13.3.3　Boot Loader

如前所述，在启动的过程中，BIOS 会把 Boot Loaer 读入内存，并把控制权交给它。MBR（硬盘启动）或软盘的启动扇区（软盘启动）内的代码就是 Boot Loader 或者 Boot Loader 的一部分。Boot Loader 的实现是很复杂的，这也是我们什么要了解磁盘结构的原因。

实际上 Boot Loader 的来源有多种，最常见的一种是你的操作系统就是 DOS，而 Boot

Loader 是 DOS 系统提供的 MS-Boot Loader。这种情况下比较简单：如果是软盘启动，Boot Loader 会检查盘上是否存在两个隐含的系统文件（IBMBIO.COM、IBMDOS.COM），若有，读出并送至内存中指定的区域，把控制权转移给 IBMBIO 这个模块，否则显示出错信息。如果是硬盘启动，Boot Loader 将查找主分区表中标记为活动分区的表项，把该表项对应的分区的引导扇区读入，然后把控制权交给该扇区内的引导程序，这段程序也可以被看作是 Boot Loader 的一部分，它完成的工作与软盘的 Boot Loader 大致相同。

有时候一台计算机上所装的操作系统并不是 DOS，或者并不仅仅是 DOS。在这种情况，如果你是一个 Linux 的使用者，那么，你的计算机上现在就需要两套操作系统了（Linux 和 Windows），于是启动碰上了一个新问题——怎么能引导多个系统?

仅仅 MS-DOS 的 Boot Loader 无法完成这种工作，你所要的是一个可以多重启动的工具，怎么办？幸运的是，有很多用来实现这一功能的软件（大部分是共享或自由软件）已经被编制了，如在 DOS 环境下启动 Linux 的 LOADLIN，Linux 下最常用的 LILO 等等，使用它们，你可以方便地从各种操作系统启动。不幸的是，你是一个操作系统的研究者，很可能你会需要编写自己的 Boot Loader 程序，所以，你必须了解这个程序的工作原理。由于 LILO 的强大功能和方便使用的特性，很多人都在使用它，因此我们将在这里详细介绍 LILO。

13.3.4 LILO

LILO 是一个在 Linux 环境编写的 Boot Loader 程序(所以安装和配置它都要在 Linux 下)。它的主要功能就是引导 Linux 操作系统的启动。但是它不仅可以引导 Linux，它还可以引导其他操作系统，如 DOS，Windows 等等。它不但可以作为 Linux 分区的引导扇区内的启动程序，还可以放入 MRB 中完全控制 Boot Loadr 的全过程。下面让我们看看几种典型情况下硬盘的主引导扇区和各个分区的引导扇区内程序的内容。

（1）计算机上只装了 DOS 一个操作系统

这种情况和刚才介绍的 DOS 硬盘启动相对应，如图 13.6 所示。



图 13.6　只有 DOS 的硬盘分区图

（2）计算机上装了 DOS 和 Linux 操作系统，Linux 由 LOADIN 启动，如图 13.7 所示。

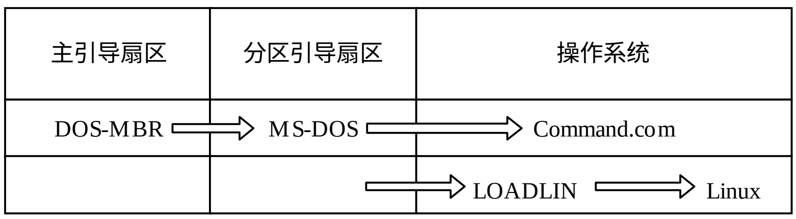


图 13.7　用 LOADLIN 从 DOS 下启动

Linux 在这种情况下，DOS 的主引导区没有发生变化，分区的引导扇区也没有变化，只是在 DOS 的配置文件 Autoexec.bat 中加入了 Loadin 程序而已。

（3）LILO 存放在 Linux 分区的引导扇区内，如图 13.8 所示。

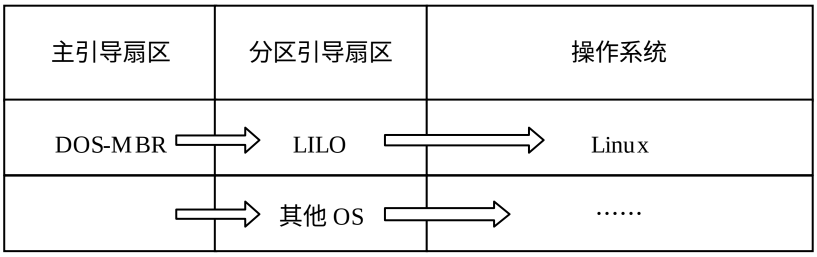


图 13.8　LILO 存放在 Linux 分区的引导扇区内

在这种情况下，LILO 存放在硬盘上的一个基本分区内。如果希望从 Linux 启动，必须把 Linux 分区设为活动分区。而如果想使用 Windows，就必须把 Windows 所在的分区激活，然后重新启动以进入 Windows，也就是说，你没办法在启动的时候选择从哪个操作系统进入，这样的多重启动显得太麻烦。回想 Windows 下的多重启动，你只要在引导时输入一个 F3 键，便能自动进入 DOS 6.22，这才是我们所希望的方式。Windows 能做到，LILO 当然可以做到，它还能做得更好(LILO 不仅允许你选择从哪个系统引导，它还允许你给 Linux 的内核传递参数)请看下面这种模式，如图 13.9 所示。

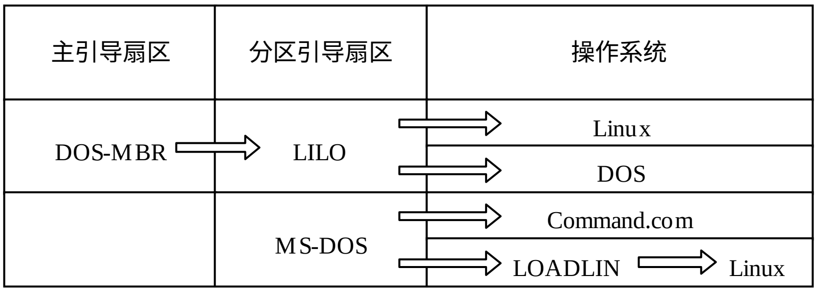


图 13.9　LILO 在分区引导扇区内的多重引导

无论从哪个分区引导，你都可以选择是进入 Linux 或是 DOS，不过从 DOS 分区启动时，如果你不想进入 Linux，你需要单步执行 autoexec.bat 以跳过 LOADLIN。而从 Linux 分区启动时，你仅需要在启动时敲键盘输入操作系统的名字（这个名字可以由你在配置 LILO 时自己设定）便可以选择进入哪个操作系统。这张表完全是用于说明 LILO 安装位置的，其实你可能已经看出来了，既然无论从哪个分区都能进入所有的操作系统，那么，只要有一个活动分区就够了，从方便的角度来讲，从 Linux 分区启动是个不错的选择。

（4）LILO 放在硬盘的主引导扇区里。

如图 13.10 所示。LILO 如果在安装时选择作为 MRB，它将负责 Boot Loader 的全过程，不过这样做有一定的风险，因为它将覆盖 MBR，有可能使你原来的系统无法启动，所以你需

要先备份主引导扇区。

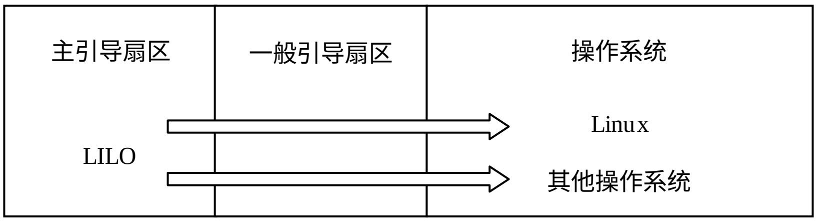


图 13.10　LILO 放在主引导扇区里

LILO 的功能实际上是由几个程序共同实现的，它们是：

① Map Installer：这是 LILO 用于管理启动文件的程序。它可以将 LILO 启动时所需的文件放置到合适的位置(这些文件的位置由 LILO 本身决定)并且记录下这些位置，以便 LILO 访问。其实，当运行/sbin/lilo 这个程序时，Map installer 就已经工作了，它将 Boot loader 写入引导分区（原来的 Boot Loader 将被备份），创建记录文件——map file 以映射内核的启动文件。每当内核发生变化时（比如说内核升级了），你必须运行/sbin/lilo 来保证系统的正常运行。

② Boot Loader：这就是由 BIOS 读入内存的那部分 LILO 的程序，它负责把 Linux 的内核或其他操作系统的引导分区读入内存。另外，Linux 的 Boot Loader 还提供一个命令行接口，可以让用户选择从哪个操作系统启动和加入启动参数。

③ 其他文件：这些文件主要包括用于存放 Map installer 记录的 map 文件（/boot/map）和存放 LILO 配置信息的配置文件（/etc/lilo.conf），这些文件都是 LILO 启动时必需的，它们一般存放在/boot 目录下。

LILO 在引导 Linux 的同时还可以向 Linux 的内核传送参数。前面我们提到了，LILO 提供了一个命令行解释程序，当系统加载 LILO，并在屏幕上显示了“LILO” 字样时，你可以按下 Ctrl 或者 Shift 键（不同版本的 LILO 可能有所不同，笔者的系统需要按下 Tab 键），这时会出现“LILO boot”字样，表明命令行解释程序已经被激活，可以从键盘输入了。如果相应的系统引导提示符是“Linux”，“DOS”的话，你可以键入“Linux”启动 Linux，或者键入“DOS”启动 Windows。如果选择启动 Linux，此时你还可以在“Linux”后面加入一些参数，LILO 可以把这些参数传递给内核。例如：

LILO boot: Linux 1 告诉内核按照单用户模式启动。

LILO boot: Linux ether=eth0,0x280,10 告诉内核你的第一块网卡的端口地址是 0x280，中断号是 10。

LILO 提供许多种参数，如 Debug，等等，具体这些参数和它们的作用，请你查阅 LILO 的文挡。此外，并不是所有的硬件都需要加参数才能支持的。如果硬件设备在编译内核时已经被支持了，那么完全没有必要加参数。事实上，只有那些比较特殊（也比较不常用）的设备，才需要在启动时设定参数值，明确它的端口地址和中断号，以节省大量的用于检测端口地址和中断号的启动时间。

像可以预设默认的启动选项一样，在/etc/lilo.conf 中也可以预先定义启动时要输入的参数，这样就可以避免每次启动都要重复输入。让我们看一个具体的 lilo.conf 的例子，例

子的左边是 Script 的脚本程序，右面是对程序的解释。从这个例子可以看出，lilo.conf 的编制思想，同 DOS 下的 config.sys 差不多。

```
# /etc/lilo.conf
# LILO configuration file
# gernerated by 'liloconfig'
#
# Start LILO global section      /*LILO 的通用配置块*/
append = "ether=eth0, 0x280, 10" /*请注意，这就是向内核传递的参数，我们把它写在这里，就可以免去每次在启动时输入的麻烦*/
 boot = /dev/hda2                    /*LILO 安装在硬盘 1 的二号分区的分区表上*/
delay = 50                               /*给用户选择从哪个操作系统启动的等待时间*/
vga = normal                         /*显示器设置为标准 VGA*/
# ramdisk = 0                            /*未安装虚拟启动盘*/
# End LILO global section            /*通用配置块结束*/
# Linux bootable partition config begins      /*用于启动 Linux 的配置块*/
root = /dev/hda2                     /*Linux 的根文件系统安装在硬盘 1 的二号分区上*/
image = /vmlinuz                 /*选择根目录下的 vmlinuz 作为内核*/
lable = linux                    /*启动选择的标识符为 linux*/
image = /zimage-2.4.18           /*在引导 Linux 时，可以选择多个内核。比如说我们编译了一个新的内核，并想从它启动，只需把这行程序写在这里，当然，别忘了先运行 LILO 来改变 Map 文件。*/
label = Newkernel
read-only                        /*以只读方式安装，防止启动中的误操作*/
#Linux bootable partition config ends      /*Linux 配置块结束*/
#DOS bootable partition config begens          /*用于启动 DOS 的配置块*/
other = /dev/hda1  /*该操作系统的 Boot Loader 安装在硬盘 1 的一号分区的分区表内*/
label = dos                  /*启动选择的标识符为 dos*/
table = /dev/hda            /*该操作系统的根目录在硬盘 1 的一号分区上*/
# Dos bootab le partion config ends   /*Dos 配置块结束*/
```

13.3.5　LILO 的运行分析

我们知道了 LILO 怎么安装，包含什么东西，有什么功能，但 LILO 到底是怎么运行的呢？下面是代码分析层次的 LILO 运行过程，通过介绍这个过程，希望你能对整个 Boot Loader 这部分内容有一个深入的认识。

1．从软盘启动

Linux 内核可以存入一张 1.44MB 的软盘中，这样做的前提是对“Linux 内核映像”进行压缩，压缩是在编译内核时进行的，而解压是由装入程序在引导时进行的。

当从软盘引导 Linux 时，Boot Loader 比较简单，其代码在 arch/i386/boot/bootsect.S 汇编语言文件中。当编译 Linux 内核源码时，就获得一个新的内核映像，这个汇编语言文件所产生的可执行代码就放在内核映像文件的开始处。因此，制作一个包含 Linux 内核的软磁盘并不是一件困难的事。

把内核映像的开始处拷贝到软盘的第 1 个扇区就创建了一张启动软盘。当 BIOS 装入软盘的第 1 个扇区时，实际上就是拷贝 Boot Loader 的代码。BIOS 将 Boot Loader 读入至内存中物理地址 0x07c00 处，控制权转给 Boot Loader，Boot Loader 执行如下操作。

- 把自己从地址 0x07c00 移到 0x90000。
- 利用地址 0x03ff，建立“实模式”栈。
- 建立磁盘参数表，这个表由 BIOS 用来处理软盘设备驱动程序。
- 通过调用 BIOS 的一个过程显示“Loading”信息。
- 然后，调用 BIOS 的一个过程从软盘装入内核映像的 setup( )代码，并把这段代码放入从地址 0x90200 开始的地方。
- 最后再调用 BIOS 的一个过程。这个过程从软盘装入内核映像的其余部分，并把映像放在内存中从地址 0x10000 开始的地方，或者从地址 0x100000 开始的地方，前者叫做“低地址”的小内核映像（以“make zImage”进行的编译），后者叫做“高地址”的大内核映像（以“make bzImage”）进行的编译。

2．从硬盘启动

一般情况下，Linux 内核都是从硬盘装入的。BIOS 照样将引导扇区读入至内存中的 0x00007c00 处，控制权转给 Boot Loader，Boot Loader 把自身移动至 0x90000 处，并在 0x9B000 处建立堆栈（从 0x9B000 处向 0x9A200 增长），将第 2 级的引导扇区读入至内存的 0x9B000 处，把控制权交给它。在引导扇区移动之后，将显示一个大写的 L 字符，而在启动第 2 级的引导扇区之前，将显示一个大写的 I 字符。如果读入第 2 级的引导扇区的过程有错误，屏幕上的 LI 之后会显示一个十六进制的错误号。

二级引导扇区内的代码将把描述符表读入至内存中的 0x9D200 处，把包含有命令行解释程序的扇区读入至内存的 0x9D600 处。接着，二级引导扇区将等待用户的输入，不管这时用户输入了一个选择还是使用缺省配置，都将把对应的扇区读入至内存的 0x9D600（覆盖命令行解释程序的空间），把生成的启动参数保存在 0x9D800 处。

如果用户定义了用于启动的 RAM 盘的话，这部分文件将被读入到物理内存的末尾。如果你的内存大于 16MB 的话，它会被读入至 16MB 内存的结尾，这是因为 BIOS 程序不支持对 16MB 以上内存的访问（它用于寻址的指令中只有 24 位的地址描述位）。并且它开始于一个新的页，以便于启动后系统把它所占的内存回收到内存池。

接下来，操作系统的初始化代码将被读入到内存的 0x90200 处。而系统的内核将被读入到 0x10000 处。如果该内核是以 make bzImage 方式编译的，它将被读入到内存的 0x100000 处。在读入的过程中，存放 map 文件的扇区被读入至内存的 0x9D000 处。

如果读入的 image 是 Linux 的内核，控制权将交给处于 0x90200 的 Setup.S。如果读入的是另外的操作系统，过程要稍微麻烦一点:chain loader 被读入到内存的 0x90200 处。该系统用于启动的扇区被读入到 0x90400。chain loader 将把它所包含的分区表移到 0x00600 处，把引导扇区读入到 0x07c00。做完这一切，它把控制权交给引导扇区。

第 2 级引导扇区在得到控制权以后马上显示一个大写的 L 字符。读入命令行解释程序后显示一个大写的 O 字符。

图 13.11 是 LILO 运行完后，内存的分布情况。

地址	内容	大小
0x0000		1982 字节
0x007BE	分区表	64 字节
0x007FE		29K 字节
0x07C00	MBA 或软盘的引导扇区	512 字节
0x7E00		32.5K 字节
0x10000	内核	448K 字节
0x90000	软盘引导扇区	512 字节
0x90200	Linux 的 Setup.s	39.5K 字节(其中占用了 2K 字节)
0x9A000	主引导扇区	512 字节
0x9A200	堆栈	3.5K 字节
0x9B000	第 2 级引导扇区	8K 字节，占用了 2.5K 字节
0x9D000	Map 信息	512 字节
0x9D200	描述符表	1K 字节
0x9D600	命令行参数	512 字节
0x9D800	键盘信息交换区	512 字节
0x9DA00	启动参数	1K 字节
0x9DC00		7.5K 字节
0xA0000	driver swapper	1K 字节

图 13.11　LILO 运行后的计算机内存分布情况

13.4　进入操作系统

Boot Loader 作了这么多工作，一言以蔽之，只是把操作系统的代码调入内存，所以，当它执行完后，自然该把控制权交给操作系统，由操作系统的启动程序来完成剩下的工作。上面已经提到了，LILO 此时把控制权交给了 Setup.S 这段程序。该程序是用汇编语言编写的 16 位启动程序，它作了些什么呢？

13.4.1　Setup.S

首先，Setup.S 对已经调入内存的操作系统代码进行检查，如果没有错误（所有的代码都已经被调入，并放至合适的位置），它会通过 BIOS 中断获取内存容量信息，设置键盘的响应速度，设置显示器的基本模式，获取硬盘信息，检测是否有 PS/2 鼠标，这些操作，都是在 386 的实模式下进行的，这时，操作系统就准备让 CPU 进入保护模式了。当然，要先屏蔽中

断信号，否则，系统可能会因为一个中断信号的干扰而陷入不可知状态，然后再次设置 32 位启动代码的位置,这是因为虽然预先对 32 位启动程序的存储位置有规定,但是 Boot Loader (通常是 LILO)有可能把 32 位的启动代码读入一个与预先定义的位置不同的内存区域，为了保证下一个启动过程能顺利进行，这一步是必不可少的。

完成上面的工作后，操作系统指令 lidt 和 lgdt 被调用了，中断向量表（idt）和全局描述符表（gdt）终于浮出水面了，此时的中断描述符表放置的就是开机时由 BIOS 设定的那张表，gdt 虽不完善，但它也有了 4 项确定的内容，也就是说，这里已经定义了下面 4 个保护模式下的段。

```
(1) .word   0，0，0，0      ! 系统所定义的 NULL 段
(2) .word   0，0，0，0      ! 空段，未使用
(3) .word   0xFFFF  ! 4Gb  (0x100000*0x1000 = 4Gb) 大小的系统代码段
    .word   0x0000      !base address=0
    .word   0x9A00      ! 可执行代码段
    .word   0x00CF      !粒度=4096
(4) .word   0xFFFF      ! 4Gb (0x100000*0x1000 = 4Gb) 大小的系统数据段
    .word   0x0000      ! base address=0
    .word   0x9200      !可读写段
    .word   0x00CF      !粒度=4096
```

注意：这里关于段描述符的格式请看第二章图 2.10 及相关内容。

在实模式下，还有几件事要作，如下所述。

我们需要对 8259 中断控制器进行编程，当然，这很简单，让它们的功能与标准的 PC 相一致就可以了，这已经在第四章进行了介绍，在此不准备再进行详细介绍。

此外，协处理器也需要重新复位。这几件事做完以后，Setup.S 设置保护模式的标志位，重新取指令以后，再用一条跳转指令：

```
jmpi    0x100000, KERNEL_CS
```

进入保护模式下的启动阶段，同时把控制权交给 Head.S 这段纯 32 位汇编代码。

13.4.2 Head.S

Head.S 也要先做一些屏蔽中断一类的准备工作,然后,它会对中断向量表做一定的处理：用一个默认的表项把所有的 256 个中断向量填满。这个默认表项指向一个特殊的中断服务程序，事实上，该程序什么都不做。为什么这样呢？这是因为，在 Linux 系统初始化完成后，BIOS 的中断服务程序是不会再被使用了。Linux 采用了很完善的设备驱动程序使用机制，该机制使特定硬件设备的中断服务程序很容易被系统本身或用户直接调用，而且，调用时所需的参数通常都要比 BIOS 调用来得简单。由于设备驱动程序是专门针对一个设备的，所以，它所包含的中断服务程序在功能上通常也比 BIOS 中断要完善，所以，BIOS 的中断向量在这里就被覆盖了。启动处在这个阶段，第一，还不需要开启中断，第二，相应的设备驱动程序还未被加载，所以把中断向量表置空显然是个合理的选择。事实上，直到初始化到了最后的阶段（start_kernel()被调用后），中断向量表才被各个中断服务程序重新填充。

Boot Loader 读入内存中的启动参数和命令行参数，这些参数在启动过程中是必须的数据资料，他们不但在启动的过程中要使用，在启动后也是必须的，Head.S 把它们保存在 empty_zero_page 页中，这就涉及到了页机制，之所以先映射这个页，是因为以后运行的程序可能会占用启动参数和命令行参数的内存区，所以要先保存它们。

Head.S 此后会检查 CPU 的类型：虽然 Intel 系列 CPU 保持兼容，但无疑后继产品会有很多新的特性和功能（如奔腾芯片支持 4MB 大小的页）。作为一个操作系统，Linux 当然要对他们作出相应的支持并发挥他们更优越的处理能力，此外，由于 Intel 已成为业界标准，所以也有必要把与之兼容的设备归入一类（如 AMDK6 就与 Intel 奔腾芯片兼容）。这里只是对处理器类型进行判断，在 Start_kernel()中要根据这里的结果对系统进行设置。此外，还要对协处理器进行检查，80387 与 80487 当然也该区别对待。

下面是一个非常重要的初始化步骤——页初始化。

Head.S 调用了 Setup_paging 这个子函数，这个函数只对 4MB 大小的内存空间进行了映射，剩下的部分在 start_kernel()中分配，这个函数并不复杂，它先把从线性地址 0xC0000000 处开始的两个大小为 4KB 的内存空间映射为两个页：一个为页目录表——swap_page_dir，另一个为零页——pg0，他们分别指向物理地址的 0x1000 和 0x2000。此后设置页目录表的属性（可读写、存在、任意特权级可访问），并使它的表项指向为内核分配的 4MB 空间上的各个页。在 Head.S 里定义了几个比较特殊的页，除了上面提到的 swap_page_dir，pg0 以外，还有 empty_bad_page、empty_bad_page_table、empty_zero_page 等等。刚才提到了 empty_zero_page，这里详细介绍一下这个页，以便对页机制及页初始化有更进一步的认识。

empty_zero_page 这个页存放的是系统启动参数和命令行参数，他们各占 2KB 大小，即各占半页。该页的具体内容如下：

```
偏移量     数据类型            描述
------    ----       -----------
   0      32 bytes       一段屏幕显示信息
2    unsigned shortEXT_MEM_K,  extended memory size in Kb (BIOS 的 15 号
中断得到的内存大小)
 0x20     unsigned shortCL_MAGIC,  命令行标志数 (=0xA33F)
 0x22     unsigned shortCL_OFFSET,  经计算所得的命令行程序的偏移地址，
                 0x90000 + contents of CL_OFFSET
               (只在 CL_MAGIC = 0xA33F 时计算)
 0x40     20 bytes  APM_BIOS_INFO 结构体
 0x80     16 bytes  hd0-disk-parameter  (硬盘 1 参数，由 BIOS 的 40 号中断得到)
 0x90     16 bytes  hd1-disk-parameter  (硬盘 1 参数，由 BIOS 的 46 号中断得到)

 0xa0     16 bytes       系统描述符表截为 16 字节的信息
               ( sys_desc_table_struct 结构)
 0xb0 - 0x1df       未占用
0x1e0     unsigned long ALT_MEM_K，可变化的内存大小
0x1f1     char      setup.s 所占的扇区数
0x1f2     unsigned shortMOUNT_ROOT_RDONLY (if !=0)
0x1f4     unsigned short压缩内核占用的空间
0x1f6     unsigned shortswap_dev (unused AFAIK) (交换分区)
0x1f8     unsigned shortRAMDISK_FLAGS (虚盘的标志)
0x1fa     unsigned shortVGA-Mode
0x1fc     unsigned shortORIG_ROOT_DEV (high=Major,  low=minor) (主设备的从
```

设备号）

```
0x1ff     char      AUX_DEVICE_INFO
0x200     short          jump to start of setup code aka "reserved" field.
0x202     4 bytes        SETUP-header 的标志， ="HdrS"
               目前的版本是 0x0241...
0x206     unsigned shortheader 的标志
0x208     8 bytes         setup.S 获得 boot loaders 信息的内存区
0x210     char      LOADER_TYPE = 0， 老式引导程序，
               否则由 boot loader 自己设置
               0xTV: T=0 ： LILO
                    1 ： Loadlin
                    2 ： bootsect-loader
                    3 ： SYSLINUX
                    4 ： ETHERBOOT
                    V = 引导程序版本号
0x211     char      loadflags（调入标志）:
               bit0 = 1: 读入高内存区 (bzImage)
               bit7 = 1: boot loader 设置了堆和指针。
0x212     unsigned short(setup.S)
0x214     unsigned long KERNEL_START， loader 从何处调入内核
0x218     unsigned long INITRD_START， 调入内存的 image 的位置
0x21c     unsigned long INITRD_SIZE， 虚盘上的 image 的大小
0x220     4 bytes        (setup.S)
0x224     unsigned shortsetup.S 的堆和指针
0x226 - 0x7ff      setup.S 的程序体

0x800     string， 2K max     命令行参数
```

这个页中的内容是从启动的多个程序中总结出来的，从中我们可以看出，页的大小为 0x800～4KB，页内的数据按照一定的偏移量顺序排列，对页的读写操作也要通过这些偏移量来完成。在你分析 Linux 的启动时，通过和这张表的对照，希望你能对在这种汇编语言环境中准确的定位读写及对页与段的安排，有一个更明确的认识。

此外，你应该发现，物理地址的 0x0000 没有被页映射，这部分空间保存着中断向量表，全局描述符表等系统的比较重要的数据结构，他们不必要进行页交换，而且，对线性空间的访问页机制也完全不能察觉，所以，没有必要用页来映射这个空间。这样做还有另外一层用意：系统中凡是无效的指针（NULL），都可以自动对应到这个空间。

关于页初始化更详细的内容请参见第六章。那么，段机制在 Head.S 中有没有变化呢？我们知道，一旦系统进入保护模式，那么，系统的多任务特性就完全可以体现了。这里，段机制的多任务特性，当然要进一步地体现出来了。下面是 Head.S 中定义的全局描述符表：

```
ENTRY(gdt_table)
     .quad 0x0000000000000000        /* NULL descriptor */
     .quad 0x0000000000000000        /* not used */
     .quad 0x00cf9a000000ffff        /* 0x10 kernel 4GB code at 0x00000000 */
     .quad 0x00cf92000000ffff        /* 0x18 kernel 4GB data at 0x00000000 */
     .quad 0x00cffa000000ffff        /* 0x23 user   4GB code at 0x00000000 */
     .quad 0x00cff2000000ffff        /* 0x2b user   4GB data at 0x00000000 */
     .quad 0x0000000000000000        /* not used */
     .quad 0x0000000000000000        /* not used */
     /*
```

```
 * The APM segments have byte granularity and their bases
 * and limits are set at run time.
 */
.quad 0x0040920000000000        /* 0x40 APM set up for bad BIOS's */
.quad 0x00409a0000000000        /* 0x48 APM CS    code */
.quad 0x00009a0000000000        /* 0x50 APM CS 16 code (16 bit) */
.quad 0x0040920000000000        /* 0x58 APM DS    data */
.fill NR_CPUS*4,8,0              /* space for TSS's and LDT's */
```

关于这部分内容的详细解释请参见 2.3 节。有了新的全局描述符表和中断向量表，当然要重新装入描述符，所以 lgdt，lidt，又被执行了一遍，而以前预取的指令当然要重取，各个寄存器也要重新赋值，核心的堆栈自然也要重置。此外，虽然此时并没有用户以和任务，但还是可以用 lldt 让局部描述符表的寄存器指向空，以便初始化的顺利进行。

到这一步，保护机制下内存管理，中断管理的框架已经建好了，下一步，就是怎么样具体实现操作系统的功能了，于是，Head.S 调用/init/main.C 中的 start_kernel 函数，把控制权交给启动的下一部分代码。

13.5　main.c 中的初始化

head.s 在最后部分调用 main.c 中的 start_kernel（）函数，从而把控制权交给了它。所以启动程序从 start_kernel（）函数继续执行。这个函数是 main.c 乃至整个操作系统初始化的最重要的函数，一旦它执行完了，整个操作系统的初始化也就完成了。

如前所述，计算机在执行 start_kernel（）前处已经进入了 386 的保护模式，设立了中断向量表并部分初始化了其中的几项，建立了段和页机制，设立了 9 个段，把线性空间中用于存放系统数据和代码的地址映射到了物理空间的头 4MB，可以说我们已经使 386 处理器完全进入了全面执行操作系统代码的状态。但直到目前为止，我们所做的一切可以说都是针对 386 处理器所做的工作，也就是说几乎所有的多任务操作系统只要使用 386 处理器，都需要作这一切。而一旦 start_kernel（）开始执行，Linux 内核的真实面目就一步步地展现在你的眼前了。start_kernel（）执行后，你就可以以一个用户的身份登录和使用 Linux 了。

让我们来看看 start_kernel 到底做了些什么，这里，我们通过介绍 start_kernel（）所调用的函数，来讨论 start_kernel（）的流程和功能。

我们仿照 C 语言函数的形式来进行这种描述，不过请注意，真正的 start_kernel（）函数调用子函数并不象我们在下面所写的这样简单，毕竟这本书的目的是帮助你深入分析 Linux。我们只能给你提供从哪儿入手和该怎么看的建议，真正深入分析 Linux，还需要你自己来研究代码。start_kernel（）这个函数是在/init/main.c 中，这里也只是将 main.c 中较为重要的函数列举出来。

```
start_kernel ()            /*定义于 init/main.c */
{
……
setup_arch () ;
}
```

它主要用于对处理器、内存等最基本的硬件相关部分的初始化，如初始化处理器的类型

（是在 386，486，还是 586 的状态下工作，这是有必要的，比如说，Pentium 芯片支持 4MB 大小的页，而 386 就不支持），初始化 RAM 盘所占用的空间（如果你安装了 RAM 盘的话）等。其中，setup_arch（）给系统分配了 intel 系列芯片统一使用的几个 I/O 端口的地址。

paging_init（）；　/*该函数定义于 arch/i386/mm/init.c */

它的具体作用是把线性地址中尚未映射到物理地址上的部分通过页机制进行映射。这一部分在本书第六章有详细的描述，在这里需要特别强调的是，当 paging_init（）函数调用完后，页的初始化就整个完成了。

trap_init（）；/*该函数在 arch/i386/kernel/traps.c 中定义*/

这个初始化程序是对中断向量表进行初始化，详见第四章。它通过调用 set_trap_gate（或 set_system_gate 等）宏对中断向量表的各个表项填写相应的中断响应程序的偏移地址。

事实上，Linux 操作系统仅仅在运行 trap_init（）函数前使用 BIOS 的中断响应程序（我们这里先不考虑 V86 模式）。一旦真正进入了 Linux 操作系统，BIOS 的中断向量将不再使用。对于软中断，Linux 提供一套调用十分方便的中断响应程序，对于硬件设备，Linux 要求设备驱动程序提供完善的中断响应程序,而调用使用多个参数的 BIOS 中断就被这些中断响应程序完全代替了。

另外，在 trap_init（）函数里，还要初始化第一个任务的 LDT 和 TSS，把它们填入 Gdt 相应的表项中。第一个任务就是 init_task 这个进程，填写完后，还要把 init_task 的 TSS 和 LDT 描述符分别读入系统的 TSS 和 LDT 寄存器。

init_IRQ（）　/* 在 arch/i386/kernel/irq.c 中定义*/

这个函数也是与中断有关的初始化函数。不过这个函数与硬件设备的中断关系更密切一些。

我们知道 intel 的 80386 系列采用两片 8259 作为它的中断控制器。这两片级连的芯片一共可以提供 16 个引脚，其中 15 个与外部设备相连，一个用于级连。可是，从操作系统的角度来看，怎么知道这些引脚是否已经使用;如果一个引脚已被使用，Linux 操作系统又怎么知道这个引脚上连的是什么设备呢?在内核中，同样是一个数组（静态链表）来纪录这些信息的。这个数组的结构在 irq.h 中定义：

```
struct irqaction {
        void (*handler) (int, void *, struct pt_regs *) ;
           unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
   struct irqaction *next;}
```

具体内容请参见第四章。我们来看一个例子：

```
static void math_error_irq (int cpl, void *dev_id, struct pt_regs *regs)
    {
        outb (0, 0xF0) ;
        if (ignore_irq13 || !hard_math)
            return;
        math_error () ;
    }
static struct  irqaction  irq13 = { math_error_irq, 0, 0, "math error", NULL, NULL };
```

该例子就是这个数组结构的一个应用，这个中断是用于协处理器的。在 init_irq（）这

个函数中，除了协处理器所占用的引脚，只初始化另外一个引脚，即用于级连的 2 引脚。不过，这个函数并不仅仅做这些，它还为两片 8259 分配了 I/O 地址，对应于连接在管脚上的硬中断，它初始化了从 0x20 开始的中断向量表的 15 个表项（386 中断门），不过，这时的中断响应程序由于中断控制器的引脚还未被占用，自然是空程序了。当我们确切地知道了一个引脚到底连接了什么设备，并知道了该设备的驱动程序后，使用 setup_x86_irq 这个函数填写该引脚对应的 386 的中断门时，中断响应程序的偏移地址才被填写进中断向量表。

sched_init（） /*在/kernel/sched.c 中定义*/

看到这个函数的名字可能令你精神一振，终于到了进程调度部分了，但在这里，你非但看不到进程调度程序的影子，甚至连进程都看不到一个，这个程序是名副其实的初始化程序：仅仅为进程调度程序的执行做准备。它所做的具体工作是调用 init_bh 函数（在 kernel/softirq.c 中）把 timer、tqueue、immediate 三个任务队列加入下半部分的数组。

time_init（）/*在 arch/i386/kernel/time.c 中定义*/

时间在操作系统中是个非常重要的概念。特别是在 Linux、UNIX 这些多任务的操作系统中它更是作为主线索贯穿始终，之所以这样说，是因为无论进程调度（特别是时间片轮转算法）还是各种守护进程（也可以称为系统线程，如页表刷新的守护进程）都是根据时间运作的。可以说，时间是他们运行的基准。那么，在进程和线程没有真正启动之前，设定系统的时间就是一件理所当然的事情了。

我们知道计算机中使用的时间一般情况下是与现实世界的时间一致的。当然，为了避开 CIH，把时间跳过每月 26 号也是种明智的选择。不过如果你在银行或证交所工作，你恐怕就一定要让你计算机上的时钟与挂在墙上的钟表分秒不差了。还记得 CMOS 吗？计算机的时间标准也是存在那里面的。所以，我们首先通过 get_cmos_time（）函数设定 Linux 的时间，不幸的是，CMOS 提供的时间的最小单位是秒，这完全不能满足需要，否则 CPU 的频率 1 赫兹就够了。Linux 要求系统中的时间精确到纳秒级，所以，我们把当前时间的纳秒设置为 0。

完成了当前时间的基准的设置，还要完成对 8259 的一号引脚上的 8253（计时器）的中断响应程序的设置，即把它的偏移地址注册到中断向量表中去。

parse_options（）　 /*在 main.c 中定义*/

这个函数把启动时得到的参数如 debug、init 等从命令行的字符串中分离出来，并把这些参数赋给相应的变量。这其实是一个简单的词法分析程序。

console_init（） /*在 linux/drivers/char/tty_io.c 中定义*/

这个函数用于对终端的初始化。在这里定义的终端并不是一个完整意义上的 TTY 设备，它只是一个用于打印各种系统信息和有可能发生错误的出错信息的终端。真正的 TTY 设备以后还会进一步定义。

kmalloc_init（） /*在 linux/mm/kmalloc.c 中定义*/

kmalloc 代表的是 kernel_malloc 的意思，它是用于内核的内存分配函数。而这个针对 kmalloc 的初始化函数用来对内存中可用内存的大小进行检查，以确定 kmalloc 所能分配的内存的大小。所以，这种检查只是检测当前在系统段内可分配的内存块的大小，具体内容参见第六章内存分配与回收一节。

下面的几个函数是用来对 Linux 的文件系统进行初始化的，为了便于理解，这里需要把 Linux 的文件系统的机制稍做介绍。不过，这里是很笼统的描述，目的只在于使我们对初始化的解释工作能进行下去，详细内容参见第八章的虚拟文件系统。

虚拟文件系统是一个用于消灭不同种类的实际文件系统间（相对于 VFS 而言，如 ext2、fat 等实际文件系统存在于某个磁盘设备上）差别的接口层。在这里，您不妨把它理解为一个存放在内存中的文件系统。它具体的作用非常明显：Linux 对文件系统的所有操作都是靠 VFS 实现的。它把系统支持的各种以不同形式存放于磁盘上或内存中（如 proc 文件系统）的数据以统一的形式调入内存，从而完成对其的读写操作。（Linux 可以同时支持许多不同的实际文件系统，就是说，你可以让你的一个磁盘分区使用 Windows 的 FAT 文件系统，一个分区使用 UNIX 的 SYS5 文件系统，然后可以在这两个分区间拷贝文件）。为了完成以及加速这些操作，VFS 采用了块缓存，目录缓存（name_cach）、索引节点（inode）缓存等各种机制，以下的这些函数，就是对这些机制的初始化。

inode_init（） /*在 Linux/fs/inode.c 中定义*/

这个函数是对 VFS 的索引节点管理机制进行初始化。这个函数非常简单：把用于索引节点查找的哈希表置入内存，再把指向第一个索引节点的全局变量置为空。

name_cache_init（） /*在 linux/fs/dcache.c 中定义*/

这个函数用来对 VFS 的目录缓存机制进行初始化。先初始化 LRU1 链表，再初始化 LRU2 链表。

Buffer_init（）/*在 linux/fs/buffer.c 中定义*/

这个函数用来对用于指示块缓存的 buffer free list 初始化。

mem_init（） /* 在 arch/i386/mm/init.c 中定义*/

启动到了目前这种状态，只剩下运行/etc 下的启动配置文件。这些文件一旦运行，启动的全过程就结束了，系统也最终将进入我们所期待的用户态。现在，让我们回顾一下，到目前为止，我们到底做了哪些工作。

其实，启动的每一个过程都有相应的程序在屏幕上打印与这些过程相应的信息。我们回顾一下这些信息，整个启动的过程就一目了然了。

当然，你的计算机也许速度很快，你甚至来不及看清这些信息，系统就已经就绪，即“Login：”就已经出现了。不要紧，登录以后，你只要打一条 dmesg | more 命令，所有这些信息就会再现在屏幕上。

【Loading ……】出自 bootsect.S ，表明内核正被读入。

【uncompress ……】很多情况下，内核是以压缩过的形式存放在磁盘上的，这里是解压缩的过程。

下面这部分信息是在 main.c 的 start_kernel 函数被调用时显示的。

【Linux version 2.2.6 （root@lance） （gcc version 2.7.2.3）】Linux 的版本信息和编译该内核时所用的 gcc 的版本。

【Detected 199908264 Hz processor】调用 init_time（）时打出的信息。

【Console:colour VGA+ 80x25，1 virtaul console（max 63）】调用 console_init（）打出的信息。初始化的终端屏幕使用彩色 VGA 模式，最大可以支持 63 个终端。

【Memory: 63396k/65536k available（848k kernel code，408k reserved， 856k data）】调用 init_mem（）时打印的信息。内存共计 65536KB，其中空闲内存为 63396KB，已经使用的内存中，有 848KB 用于存放内核代码，404KB 保留，856KB 用于内核数据。

【VFS:Diskquotas version dquot_6.4.0 initialized】调用 dquote_init（）打出的信息。quota 是用来分配用户磁盘定额的程序。关于这个程序请参看第八章。

以下是对设备的初始化：

【PCI: PCI BIOS revision 2.10 entry at 0xfd8d1 |
PCI: Using configuration type 1 |
PCI: Probing PCI hardware 】调用 pci_init（）函数时显示的信息。

【Linux NET4.0 for Linux 2.2
Based upon Swansea University Computer Society NET3.039
NET4: UNIX domain sockets 1.0 for Linux NET4.0.
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP，UDP，TCP】调用 socket_init（）函数时打印的信息。使用 Linux 的 4.0 版本的网络包，采用 sockets 1.0 和 1.0 版本的 TCP/IP 协议，TCP/IP 协议中包含有 ICMP、UDP、TCP 三组协议。

【Detected PS/2 Mouse Port。
Sound initialization started
Sound initialization complete
Floppy drive (s) : fd0 is 1.44M
Floppy drive (s) : fd0 is 1.44M
FDC 0 is a National Semiconductor PC87306 】调用 device_setup（）函数时打印的信息。包括对 ps/2 型鼠标、声卡和软驱的初始化。

看完上面这一部分代码和与之相应的信息，你应该发现，这些初始化程序并没有完成操作系统的各个部分的初始化，比如说，文件系统的初始化只是初始化了几个内存中的数据结构，而更关键的文件系统的安装还没有涉及，其实，这是在 init 进程建立后完成的。下面，就是 start_kernel（）的最后一部分内容。

13.6　建立 init 进程

在完成了上面所有的初始化工作后，Linux 的运行环境已经基本上完备了。此时，Linux 开始逐步建立进程了。

13.6.1　init 进程的建立

Linux 将要建立的第一个进程是 init 进程，建立该进程是以调用 kernel_thread（init，NULL，0）这个函数的形式进行的。init 进程是很特殊的——它是 Linux 的第 1 个进程，也是其他所有进程的父进程。让我们来看一下它是怎样产生的。

在调用 kernel_thread（init，NULL，0）函数时，会调用 main.c 中的另外一个函数——init（）。请注意 init（）函数和 init 进程是不同的概念。通过执行 inin（）函数，系统完成了下述工作。

- 建立 dbflush、kswapd 两个新的内核线程。
- 初始化 tty1 设备。该设备对应了多个终端（concole），用户登录时，就是登录在这些终端上的。
- 启动 init 进程。Linux 首先寻找"/etc/init"文件，如果找不到，就接着找"/bin/init"文件，若仍找不到，再去找"/sbin/init"。如果仍无法找到，启动将无法进行下去。否则，

便执行 init 文件，从而建立 init 进程。

当 etc/init（假定它存在）执行时，建立好的 init 进程将根据启动脚本文件的内容创建其它必要的进程去完成一些重要的操作。

（1）文件系统检查。

（2）启动系统的守护进程。

（3）对每个联机终端建立一个“ getty ”进程。

（4）执行“ /etc/rc ”下的命令文件。

此后，“ getty ”会在每个终端上显示“ login ”提示符，以等待用户的登录。此时“ getty ”会调用“ exec ”执行“ login ”程序，“ login ”将核对用户帐户和密码，如果密码正确，“ login ”调用“ exec ”执行 shell 的命令行解释程序（当然，也可以执行 X Windows 如果用户设置了的话）。shell 接着去执行用户默认的系统环境配置脚本文件（通常是用户的 home 目录下的 profile 文件）。

init 还有另外一个任务，当某个终端或虚拟控制台上的用户注销之后，init 进程要为该终端或虚拟控制台重新启动一个 “ getty ”，以便能够让其他用户登录。这是为什么呢？你应该发现，当用户登录时，“ getty ”用的是“ exec ”而不是“ fork ”系统调用来执行“ login ”，这样，“ login ”在执行的时候会覆盖“ getty ”的执行环境（同理，用户注册成功后，“ login ”的执行环境也会被 shell 占用）。所以，如果想再次使用同一终端，必须再启动一个“ getty ”。

此外，init 进程还负责管理系统中的“ 孤儿 ”进程。如果某个进程创建子进程之后，在子进程终止之前终止，则子进程成为孤儿进程。init 进程负责“ 收养 ”该进程，即孤儿进程会立即成为 init 进程的子进程。这是为了保持进程树的完整性。

init 进程的变种较多，大多数 Linux 的发行版本采用 sysvinit（由 Miquel van Smoorenburg）。这是一个编译好的软件包。由于 System V 而得名。UNIX 的 BSD 版本有不同的 init，主要区别在于是否具有运行级别（关于运行级别的问题下面会有专门的描述）：System V 有运行级别，而 BSD 没有运行级别。但这种区别并不是本质的区别。

图 5.12 是上述流程的流程图。

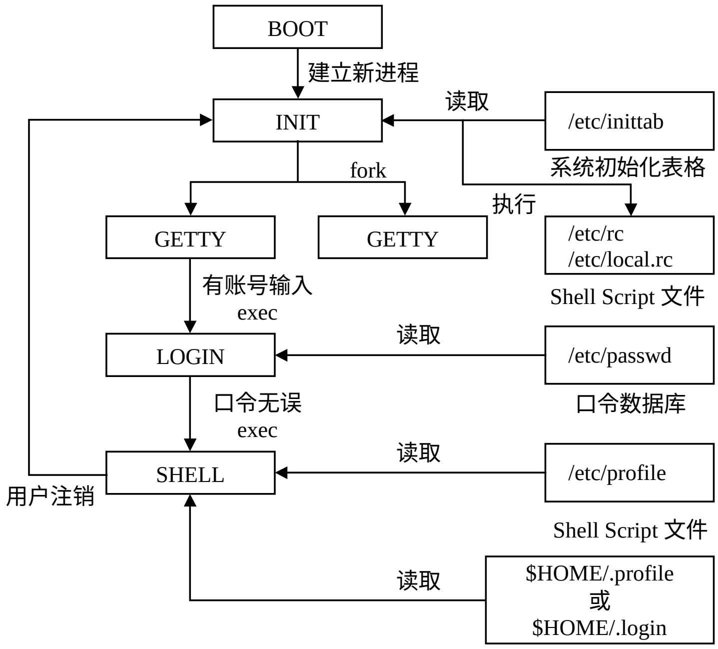


图 13.12　init 进程的启动流程

13.6.2　启动所需的 Shell 脚本文件

在启动的过程中，多次用到了 Shell 的脚本文件——Shell Script，如“$HOME/profile”、“/etc/inittab”等等。这里有必要把它们的格式和作用稍加说明。

我们把启动所需要的脚本文件分为两部分，一部分是 Linux 系统启动所必需的，也就是从/etc/inittab 开始直到出现“Login:”提示符时要用到的所用脚本，另外一部分是用户登录后自己设定的用于支持个性化的操作环境的脚本。在后者中，我们可以设定提示符用“$”或是其他什么任意你喜欢的字符，可以设所用的 Shell 是 bash、ksh，还是 zsh。显然，这部分不是我们的重点，我们要重点描述的是前一部分——系统启动所必需的脚本。

系统启动所必需的脚本存放在系统默认的配置文件目录/etc 下。用一条 ls 指令你可以看到所用的配置文件。不过，/etc 下面还有一些子目录，比如说，rc.d 就是启动中非常重要的一部分。我们主要介绍的是/etc/inittab 和 rc.d 下的一些文件，我们还是按启动时 init 进程调用它们的顺序来一一介绍。

首先调用的是/etc/inittab。init 进程将会读取它并依据其中所记载的内容进入不同的启动级别，从而启动不同的进程。所谓运行级别就是系统中定义了许多不同的级别，根据这些级别，系统在启动时给用户分配资源。比如说，以系统管理员级别登录的用户，就拥有使用几乎所有系统资源的权力，而一般用户显然不会被赋予如此大的特权。

下面是系统的 7 个启动级别。

0　系统停止。如果在启动时选择该级别，系统每次运行到 inittab 就会自动停止，无法启动。

1　单用户模式。该模式只允许一个用户从本地计算机上登录，该模式主要用于系统管理员检查和修复系统错误。

2　多用户模式。与 3 级别的区别在于用于网络的时候，该模式不支持 NFS（网络文件系统）。

3　完全多用户模式。可以支持 Linux 的所有功能，是 Linux 安装的默认选项。

4　未使用的模式。

5　启动后自动进入 X Windwos。

6　重新启动模式。如果在启动时选择该级别，系统每次运行到 inittab 就会自动重新启动，无法进入系统。

以上这 7 种模式并非在启动后一成不变，如果用户是系统管理员，那么就可以使用" init X "（X 取 1、2……6 中的一个）来改变运行状态。另外，如果重新启动时，也可以看到显示器上的提示，当前运行状态已经变为 6。

让我们看一个 inittab 文件的实例。

id:3:initdefault:　　　　　系统默认模式为 3。

#System initialization.

si::sysinit:/etc/rc.d/rc.sysinit　无论从哪个级别启动，都执行/etc/rc.d/rd.sysinit。

10:0:wait:/etc/rc.d/rc.0　　从 0 级别启动，将运行 rc.0。

11:1:wait:/etc/rc.d/rc.1　　从 1 级别启动，将运行 rc.1。

12:2:wait:/etc/rc.d/rc.2　　从 2 级别启动，将运行 rc.2。

13:3:wait:/etc/rc.d/rc.3　　从 3 级别启动，将运行 rc.3。

14:4:wait:/etc/rc.d/rc.4　　从 4 级别启动，将运行 rc.4。

15:5:wait:/etc/rc.d/rc.5　　从 5 级别启动，将运行 rc.5。

16:6:wait:/etc/rc.d/rc.6　　从 6 级别启动，将运行 rc.6。

#Things to run in every runlevel 任何级别都执行的配置文件。

ud::once:/sbin/update

#Run gettys in standard runlevels　　对虚拟终端的初始化。

1:12345:respqwn:/sbin/mingetty tty1　　tty1 运行于 1、2、3、4、5 五个级别。

2:2345:respqwn:/sbin/mingetty tty2　　tty2 运行于 2、3、4、5 四个级别。

3:2345:respqwn:/sbin/mingetty tty3　　tty3 运行于 2、3、4、5 四个级别。

4:2345:respqwn:/sbin/mingetty tty4　　tty4 运行于 2、3、4、5 四个级别。

5:2345:respqwn:/sbin/mingetty tty5　　tty5 运行于 2、3、4、5 四个级别。

6:2345:respqwn:/sbin/mingetty tty6　　tty6 运行于 2、3、4、5 四个级别。

#Run xdm in runlevel 5　　在级别 5 启动 X Window。

x:5:respawn:/usr/bin/X11/xdm -nodaemon

现在详细解释一些 inittab 的内容。

从上面的文件可以看出，inittab 的每一行分成 4 个部分，这 4 个部分的格式如下：

id:runleveld:action:process

它们代表的意义分别如下。

id：代表有几个字符所组成的标识符。在 inittab 中任意两行的标识符不能相同。

runlevels：指出本行中第 3 部分的 action 以及第 4 部分的进程会在哪些 runlevel 中被执行，这一栏的合法值有 0、1、2...…6，s 以及 S。

action：这个部分记录 init 进程在启动过程中调用进程时，对进程所采取的应答方式，合法的应答方式有下面几项。

- initdefault：指出系统在启动时预设的运行级别。上例中的第 1 行就用了这个方式。所以系统将在启动时，进入 runlevel 为 3 的模式。当然，可以把 3 改为 5，那将会执行/etc/rc.d/rc.5，也就是 X Window。
- sysinit：在系统启动时，这个进程肯定会被执行。而所有的 inittab 的行中，如果它的 action 中有 boot 及 bootwait，则该行必须等到这些 action 为 sysinit 的进程执行完之后才能够执行。
- wait：在启动一个进程之后，若要再启动另一个进程，则必须等到这个进程结束之后才能继续。
- respawn：代表这个 process 即使在结束之后，也可能会重新被启动，最典型的例子就是 getty。

明白了 inittab 的意思，让我们回过头来看看启动过程。

首先，执行的是/etc/rd.c/rc.sysinit。这里不再给出它的程序清单，只给出它的主要功能：

（1）检查文件系统

包括启用系统交换分区，检查根文件系统的情况，使用磁盘定额程序 quato（可选项），安装内核映像文件系统 proc，安装其他文件系统。

（2）设置硬件设备

设定主机名，检查并设置 PNP 设备，初始化串行接口，初始化其他设备（根据你的机器配置情况决定）。

（3）检查并载入模块

执行完 rc.sysinit 并返回 inittab 后，init 进程会根据 inittab 所设定的运行级别去执行/etc/rc.d 目录下的相应的 rc 文件。比如说运行级别为 3，相应的 rc 文件即为 rc.3。这些文件将运行不同的启动程序去初始化各个运行级别下的系统环境，这部分启动程序最重要的作用之一是启动系统的守护进程，如在 rc.3 中，就要启动 cron、sendmial 等守护进程。

做完这一步，init 进程将执行 getty 进程从而等待用户的登录，也就是说，Linux 的启动全过程已经结束了，剩下的部分，就是整个系统等待用户需求，并为用户提供服务了。

Linux 的初始化到此就结束了，回过头来看看，它确实跟我们在开始时假设的那个简单的操作系统的初始化有许多相似之处，这些是所有初始化都应该有的相似点，希望您在分析完 Linux 源代码后，能编制出自己的操作系统的初始化代码。

附录 A　Linux 内核 API

以下函数是 Linux 内核提供给用户进行内核级程序开发可以调用的主要函数。

1. 驱动程序的基本函数

类别	函数名	功能	函数形成	参数	描述
驱动程序入口和出口点	module_init	驱动程序初始化入口点	module_init (x)	x 为启动时或插入模块时要运行的函数	如果在启动时就确认把这个驱动程序插入内核或以静态形成链接，则 module_init 将其初始化例程加入到“__initcall.int”代码段，否则将用 init_module 封装其初始化例程，以便该驱动程序作为模块来使用
	module_exit	驱动程序退出出口点	module_exit (x)	x 为驱动程序被卸载时要运行的函数	当驱动程序是一个模块，用 rmmod 卸载一个模块时 module_exit（）将用 cleanup_module（）封装 clean-up 代码。如果驱动程序是静态地链接进内核，则 module_exit（）函数不起任何作用
原子和指针操作	atomic_read	读取原子变量	atomic_read (v)	v 为指向 atomic_t 类型的指针	原子地读取 v 的值。注意要保证 atomic 的有用范围只有 24 位
	atomic_set	设置原子变量	atomic_set (v, i)	v 为指向 atomic _t 类型的指针，i 为待设置的值	原子地把 v 的值设置为 i。注意要保证 atomic 的有用范围只有 24 位
	atomic_add	把整数增加到原子变量	void atomic_add (int i, atomic_t * v)	i 为要增加的值，v 为指向 atomic_t 类型的指针	原子地把 i 增加到 v。注意要保证 atomic 的有用范围只有 24 位
	atomic_sub	减原子变量的值	void atomic_sub (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针。	原子地从 v 减取 i。注意要保证 atomic 的有用范围只有 24 位。
	atomic_sub_and_test	从变量中减去值，并测试结果	int atomic_sub_and_test (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针	原子地从 v 减取 i 的值，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位

	atomic_inc	增加原子变量的值	void atomic_inc (atomic_t * v)	v 为指向 atomic _t 类型的指针	原子地从 v 减取 1。注意要保证 atomic 的有用范围只有 24 位

续表

类别	函数名	功能	函数形成	参数	描述
原子和指针操作	atomic_dec	减取原子变量的值	void atomic_dec (atomic_t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 增加 1。注意要保证 atomic 的有用范围只有 24 位
	atomic_dec _and_test	减少和测试	int atomic_dec_ and_test (atomic_ t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 减取 1，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位
	atomic_inc _and_test	增加和测试	int atomic_ inc _and_test (atomic _t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 增加 1，如果结果为 0，则返回真；其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位
	atomic_add _negative	如果结果为负数，增加并测试	int atomic_add_ negative (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针	原子地给 v 增加 i，如果结果为负数，则返回真；如果结果大于等于 0，则返回假。注意要保证 atomic 的有用范围只有 24 位
	et_unalign ed	从非对齐位置获取值	get_unaligned (ptr)	ptr 指向获取的值	这个宏应该用来访问大于单个字节的值，该值所处的位置不在按字节对齐的位置，例如从非 u16 对齐的位置检索一个 u16 的值。注意，在某些体系结构中，非对齐访问要化费较高的代价
	put_unalig ned	把值放在一个非对齐位置	put_unaligned (val, ptr)	val 为要放置的值，ptr 指向要放置的位置	这个宏用来把大于单个字节的值放置在不按字节对齐的位置，例如把一个 u16 值写到一个非 u16 对齐的位置。注意事项同上

延时、调度及定时器例程	schedule_timeout	睡眠到定时时间到	signed long schedule_timeout (signed long timeout)	timeout 为以 jiffies 为单位的到期时间	使当前进程睡眠，直到所设定的时间到期。如果当前进程的状态没有进行专门的设置，则睡眠时间一到该例程就立即返回。如果当前进程的状态设置为： ① TASK_UNINTERRUPTIBLE：则睡眠到期该例程返回 0 ② TASK_INTERRUPTIBLE：如果当前进程接收到一个信号，则该例程就返回，返回值取决于剩余到期时间 当该例程返回时，要确保当前进程处于 TASK_RUNNING 状态

2．双向循环链表的操作

函数名	功能	函数形成	参数	描述
list_add	增加一个新元素	void list_add (struct list_head * new, struct list_head * head)	new 为要增加的新元素，head 为增加以后的链表头	在指定的头元素后插入一个新元素，用于栈的操作
list_add_tail	增加一个新元素	void list_add_tail (struct list_head * new, struct list_head * head);	new 为要增加的新元素，head 为增加以前的链表头	在指定的头元素之前插入一个新元素，用于队列的操作
list_del	从链表中删除一个元素	void list_del (struct list_head * entry);	entry 为要从链表中删除的元素	
list_del_init	从链表删除一个元素，并重新初始化链表	void list_del_init (struct list_head * entry)	entry 为要从链表中删除的元素	
list_empty	测试一个链表是否为空	int list_empty (struct list_head * head)	head 为要测试的链表	
list_splice	把两个链表合并在一起	void list_splice (struct list_head * list, struct list_head * head)	list 为新加入的链表，head 为第一个链表	
list_entry	获得链表中元素的	list_entry (ptr, type, member)	ptr 为指向 list_head 的指针，type 为一个结构体，而	

	结构		member 为结构 type 中的一个域，其类型为 list_head	
list_for_each	扫描链表	list_for_each (pos, head)	pos 为指向 list_head 的指针，用于循环计数，head 为链表头	

3．基本 C 库函数

当编写驱动程序时，一般情况下不能使用 C 标准库的函数。Linux 内核也提供了与标准库函数功能相同的一些函数，但二者还是稍有差别。

类别	函数名	功能	函数形成	参数	描述
字符串转换	simple_strtol	把一个字符串转换为一个有符号长整数	long simple_strtol (con- st char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	simple_strtoll	把一个字符串转换为一个有符号长长整数	long long simple_strtoll (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	

续表

类别	函数名	功能	函数形成	参数	描述
字符串转换	simple_strtoul	把一个字符串转换为一个无符号长整数	long long simple_strtoul (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	simple_strtoull	把一个字符串转换为一个无符号长长整数	long long simple_ strt- oull (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	vsnprintf	格式化一个字符串，并把它放在缓存中	int vsnprintf (char * buf, size_t size, const char * fmt, va_list args)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为要使用的格式化字符串，args 为格式化字符串的参数	

	snprintf	格式化一个字符串，并把它放在缓存中	int snprintf (char * buf, size_t size, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数	
	vsprintf	格式化一个字符串，并把它放在缓存中	int vsprintf (char * buf, const char * fmt, va_list args)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为要使用的格式化字符串，args 为格式化字符串的参数	
	sprintf	格式化一个字符串，并把它放在缓存中	int sprintf (char * buf, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数	
字符串操作	strcpy	拷贝一个以 NUL 结束的字符串	char * strcpy (char * dest, const char * src)	dest 为目的字符串的位置，src 为源字符串的位置	
	strncpy	拷贝一个定长的、以 NUL 结束的字符串	char * strncpy (char * dest, const char * src, size_t count)	dest 为目的字符串的位置，src 为源字符串的位置，count 为要拷贝的最大字节数	与用户空间的 strncpy 不同，这个函数并不用 NUL 填充缓冲区，如果与源串超过 count，则结果以非 NUL 结束

续表

类别	函 数 名	功　能	函数形成	参　数	描　述
字符串操作	strcat	把一个以 NUL 结束的字符串添加到另一个串的末尾	char * strcat (char * dest, const char * src)	dest 为要添加的字符串，src 为源字符串	
	strncat	把一个定长的、以 NUL 结束的字符串添加到另一个串的末尾	char * strncat (char * dest, const char * src, size_t count)	dest 为要添加的字符串，src 为源字符串，count 为要拷贝的最大字节数	注意，与 strncpy,形成对照，strncat 正常结束
	strchr	在一个字符串中查找第一次出现的某个字符	char * strchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符	

	strrchr	在一个字符串中查找最后一次出现的某个字符	char * strrchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符	
	strlen	给出一个字符串的长度	size_t strlen (const char * s)	s 为给定的字符串	
	strnlen	给出给定长度字符串的长度	size_t strnlen (const char * s, size_t count)	s 为给定的字符串	
	strpbrk	在一个字符串中查找第一次出现的一组字符	char * strpbrk (const char * cs, const char * ct)	cs 为被搜索的字符串，ct 为待搜索的一组字符	
	strtok	把一个字符串分割为子串	char * strtok (char * s, const char * ct)	s 为被搜索的字符串，ct 为待搜索的子串	注意，一般不提倡用这个函数，而应当用 strsep
	memset	用给定的值填充内存区	void * memset (void * s, int c, size_t count)	s 为指向内存区起始的指针，c 为要填充的内容，count 为内存区的大小	I/O 空间的访问不能使用 memset，而应当使用 memset_io
	bcopy	把内存的一个区域拷贝到另一个区域	char * bcopy (const char * src, char * dest, int count)	src 为源字符串，dest 为目的字符串，而 count 为内存区的大小	注意，这个函数的功能与 memcpy 相同，这是从 BSD 遗留下来的，对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio

续表

类别	函数名	功能	函数形成	参数	描述
字符串操作	memcpy	把内存的一个区域拷贝到另一个区域	void * memcpy (void * dest, const void * src, size_t count)	dest 为目的字符串，src 为源字符串，而 count 为内存区的大小	对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio
	memmove	把内存的一个区域拷贝到另一个区域	void * memmove (void * dest, const void * src, size_t count)	dest 为目的字符串，src 为源字符串，而 count 为内存区的大小	memcpy 和 memmove 处理重叠的区域，而该函数不处理
	memcmp	比较内存的两个区域	int memcmp (const void * cs, const void * ct, size_t count)	cs 为一个内存区，ct 为另一个内存区，而 count 为内存区的大小	

	memscan	在一个内存区中查找一个字符	void * memscan (void * addr, int c, size_t size)	addr 为内存区，c 为要搜索的字符，而 size 为内存区的大小	返回 c 第一次出现的地址，如果没有找到 c，则向该内存区传递一个字节
	strstr	在以 NUL 结束的串中查找第一个出现的子串	char * strstr (const char * s1, const char * s2)	s1 为被搜索的串，s2 为待搜索的串	
	memchr	在一个内存区中查找一个字符	void * memchr (const void * s, int c, size_t n)	s 为内存区，为待搜索的字符，n 为内存的大小	返回 c 第 1 次出现的位置，如果没有找到 c，则返回空
位操作	set_bit	在位图中原子地设置某一位	void set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	这个函数是原子操作，如果不需要原子操作，则调用__set_bit 函数，nr 可以任意大，位图的大小不限于一个字
	__set_bit	在位图中设置某一位	void __set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	
	clear_bit	在位图中清某一位	void clear_bit (int nr, volatile void * addr)	nr 为要清的位，addr 为位图的起始地址	该函数是原子操作，但不具有加锁功能，如果要用于加锁目的，应当调用 smp_mb__before_clear_bit 或 smp_mb__after_clear_bit 函数，以确保任何改变在其他的处理器上是可见的
	__change_bit	在位图中改变某一位	void__change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	与 change_bit 不同，该函数是非原子操作
	change_bit	在位图中改变某一位	void change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	

续表

类别	函数名	功能	函数形成	参数	描述
位操作	test_and_set_bit	设置某一位并返回该位原来的值	int test_and_set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数是原子操作

__test_and_set_bit	设置某一位并返回该位原来的值	int__test_and_set_ bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数是非原子操作，如果这个操作的两个实例发生竞争，则一个成功而另一个失败，因此应当用一个锁来保护对某一位的多个访问
test_and_clear_bit	清某一位，并返回原来的值	int test_and_clear_bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址	该函数是原子操作
__test_and_clear_bit	清某一位，并返回原来的值	int __test_and_clear_ bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址。	该函数为非原子操作
test_and_change_bit	改变某一位并返回该位的新值	int test_and_change_ bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数为原子操作
test_bit	确定某位是否被设置	int test_bit (int nr, const volatile void * addr)	nr 为要测试的第几位，addr 为位图的起始地址	
find_first_zero_bit	在内存区中查找第一个值为 0 的位	int find_first_zero_ bit (void * addr, unsigned size)	addr 为内存区的起始地址，size 为要查找的最大长度	返回第一个位为 0 的位号
find_next_zero_bit	在内存区中查找第一个值为 0 的位	int find_next_zero_ bit (void * addr, int size, int offset)	addr 为内存区的起始地址，size 为要查找的最大长度，offset 开始搜索的起始位号	
ffz	在字中查找第一个 0	unsigned long ffz (un- signed long word);	word 为要搜索的字	
ffs	查找第一个已设置的位	int ffs (int x)	x 为要搜索的字	这个函数的定义方式与 Libc 中的一样
hweight32	返回一个 N 位字的加权平衡值	hweight32 (x)	x 为要加权的字	一个数的加权平衡是这个数所有位的总和

4．Linux 内存管理中 slab 缓冲区

函数名	功能	函数形成	参数	描述
kmem_cache_create	创建一个缓冲区	kmem_cache_t * kmem_cache_create (const char * name, size_t size, size_t offset, unsigned long flags, void (*ctor) (void*, kmem_cache _t *, unsigned long), void (*dtor) (void*, kmem_cache_t *, unsigned long));	Name 为在 /proc/ slabinfo 中标识这个缓冲区的名字；size 为在这个缓冲区中创建对象的大小；offset 为页中的位移量；flags 为 slab 标志；ctor 和 dtor 分别为构造和析构对象的函数	成功则返回指向所创建缓冲区的指针，失败则返回空。不能在一个中断内调用该函数，但该函数的执行过程可以被中断。当通过该缓冲区分配新的页面时 ctor 运行，当页面被还回之前 dtor 运行
kmem_cache_shrink	缩小一个缓冲区	int kmem_cache_shrink (kmem_cache _t * cachep)	Cachep 为要缩小的缓冲区	为缓冲区释放尽可能多的 slab。为了有助于调试，返回 0 意味着释放所有的 slab
kmem_cache_destroy	删除一个缓冲区	int kmem_cache_destroy (kmem_cac- he_t * cachep);	cachep 为要删除的缓冲区	从 slab 缓冲区删除 kmem_cache_t 对象，成功则返回 0 这个函数应该在卸载模块时调用。调用者必须确保在 kmem_cache_destroy 执行期间没有其他对象再从该缓冲区分配内存
kmem_cache_alloc	分配一个对象	void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);	cachep 为要删除的缓冲区，flags 请参见 kmalloc()	从这个缓冲区分配一个对象。只有当该缓冲区没有可用对象时，才用到标志 flags
kmalloc	分配内存	void * kmalloc (size_ t size, int flags)	size 为所请求内存的字节数，flags 为要分配的内存类型	kmalloc 是在内核中分配内存常用的一个函数。flags 参数的取值如下： GFP_USER – 代表用户分配内存，可以睡眠 GFP_KERNEL – 分配内核中的内存，可以睡眠 GFP_ATOMIC – 分配但不睡眠，在中断处理程序内部使用 另外，设置 GFP_DMA 标志表示所分配的内存必须适合 DMA，例如在 i386 平台上，就意味着必须从低 16MB 分配内存

kmem_cache_free	释放一个对象	void kmem_cache_free (kmem_cache_t * cachep, void * objp)	cachep 为曾分配的缓冲区，objp 为曾分配的对象	释放一个从这个缓冲区中曾分配的对象
kfree	释放以前分配的内存	void kfree (const void * objp)	objp 为由 kmalloc () 返回的指针	

5．Linux 中的 VFS

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_invalidate	使一个目录项无效	int d_invalidate (str- uct dentry * dentry)	dentry 为要无效的目录项	如果通过这个目录项能够到达其他的目录项，就不能删除这个目录项，并返回-EBUSY。如果该函数操作成功，则返回 0
	d_find_alias	找到索引节点一个散列的别名	struct dentry * d_find_alias (struct inode *	inode 为要讨论的索引节点	如果 inode 有一个散列的别名，就获取对这个别名，并返回它，否则返回空。注意，如果 inode 是一个目录，就只能有一个别名，如果它没有子目录，就不能进行散列
	prune_dcache	裁减目录项缓存	void prune_dcache (int count)	count 为要释放的目录项的一个域	缩小目录项缓存。当需要更多的内存，或者仅仅需要卸载某个安装点（在这个安装点上所有的目录项都不使用)，则调用该函数 如果所有的目录项都在使用，则该函数可能失败
	shrink_dcache_sb	为一个超级块而缩小目录项缓存	void shrink_dcache _sb (struct super_ block * sb)	sb 为超级块	为一个指定的超级块缩小目录缓存。在卸载一个文件系统是调用该函数释放目录缓存
	have_submounts	检查父目录或子目录是否包含安装点	int have_submou- nts (struct dentry * parent)	parent 为要检查的目录项	如果 parent 或它的子目录包含一个安装点，则该函数返回真
	shrink_dcache_parent	裁减目录项缓存	void shrink_dcache _parent (struct dentry * parent)	parent 为要裁减目录项的父目录项	裁减目录项缓存以删除父目录项不用的子目录项
	d_alloc	分配一个目录项	struct dentry * d_ alloc (struct dentry * parent, const struct qstr * name)	parent 为要分配目录项的父目录项，name 为指向 qstr 结构的指针	分配一个目录项。如果没有足够可用的内存，则返回 NULL；成功则返回目录项

类别	函数名	功能	函数形成	参数	描述
	d_instantiate	为一个目录项填充索引节点信息	void d_instantiate (struct dentry * entry, struct inode * inode)	entry 为要完成的目录项，inode 为这个目录项的 inode	在目录项中填充索引节点的信息。注意，这假定 inode 的 count 域已由调用者增加，以表示 inode 正在由该目录项缓存使用
	d_alloc_root	分配根目录项	struct dentry * d_ alloc_root (struct inode * root_inode)	root_inode 为要给根分配的 inode	为给定的 inode 分配一个根("/") 目录项，该 inode 被实例化并返回。如果没有足够的内存或传递的 inode 参数为空，则返回空

续表

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_lookup	查找一个目录项	struct dentry * d_lookup (struct dentry * parent, struct qstr * name)	parent 为父目录项，name 为要查找的目录项名字的 qstr 结构。	为 name 搜索父目录项的子目录项。如果该目录项找到，则它的引用计数加 1，并返回所找到的目录项。调用者在完成了对该目录项的使用后，必须调用 d_put 释放它
	d_validate	验证由不安全源所提供的目录项	int d_validate (struct dentry * dentry, struct dentry * dparent)	dentry 是 dparent 有效的子目录项，dparent 是父目录项（已知有效）	一个非安全源向我们发送了一个 dentry，在这里，我们要验证它并调用 dget。该函数由 ncpfs 用在 readdir 的实现。如果 dentry 无效，则返回 0
	d_delete	删除一个目录项	void d_delete (struct dentry * dentry)	dentry 为要删除的目录项	如果可能，把该目录项转换为一个负的目录项，否则从哈希队列中移走它以便以后的删除
	d_rehash	给哈希表增加一个目录项	void d_rehash (struct dentry * entry)	dentry 为要增加的目录项	根据目录项的名字向哈希表增加一个目录项
	d_move	移动一个目录项	void d_move (struct dentry * dentry, struct dentry * target)	dentry 为要移动的目录项，target 为新目录项	更新目录项缓存以反映一个文件名的移动。目录项缓存中负的目录项不应当以这种方式移动

	__d_path	返回一个目录项的路径	char * __d_path (struct dentry * dentry, struct vfsmount * vfsmnt, struct dentry * root, struct vfsmount * rootmnt, char * buffer, int buflen)	dentry 为要处理的目录项，vfsmnt 为目录项所属的安装点，root 为根目录项，rootmnt 为根目录项所属的安装点，buffer 为返回值所在处，buflen 为 buffer 的长度	把一个目录项转化为一个字符串路径名。如果一个目录项已被删除，串“(deleted)”被追加到路径名，注意这有点含糊不清。返回值放在 buffer 中 “buflen”应该为页大小的整数倍。调用者应该保持 dcache_lock 锁
	is_subdir	新目录项是否是父目录项的子目录	int is_subdir (struct dentry * new_den- try, struct dentry * old_dentry)	new_dentry 为新目录项，old_dentry 为旧目录项	如果新目录项是父目录的子目录项（任何路径上），就返回 1，否则返回 0
	find_inode_number	检查给定名字的目录项是否存在	ino_t find_inode_number (struct dentry * dir, struct qstr * name)	dir 为要检查的目录，name 为要查找的名字	对于给定的名字，检查这个目录项是否存在，如果该目录项有一个 inode，则返回其索引节点号，否则返回 0

续表

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_drop	删除一个目录项	void d_drop (struct dentry * dentry)	dentry 为要删除的目录项	d_drop 从父目录项哈希表中解除目录项的哈希连接，以便通过 VFS 的查找再也找不到它。注意这个函数与 d_delete 的区别，d_delete 尽可能地把目录项表记为负的，查找时会得到一个负的目录项，而 d_drop 会使查找失败
	d_add	向哈希队列增加目录项	void d_add (struct dentry * entry, struct inode * inode)	dentry 为要增加的目录项，inode 为与目录项对应的索引节点	该函数将把目录项加到哈希队列，并初始化 inode。这个目录项实际上已在 d_alloc () 函中得到填充
	dget	获得目录项的一个引用	struct dentry * dget (struct dentry * dentry)	dentry 为要获得引用的目录项	给定一个目录项或空指针，如果合适就增加引用 count 的值。当一个目录项有引用时（count 不为 0），就不能删除这个目录项。引用计数为 0 的目录项永远也不会调用 dget

	d_unhashed	检查目录项是否被散列	int d_unhashed (struct dentry * dentry)	dentry 为要检查的目录项	如果通过参数传递过来的目录项没有用哈希函数散列过，则返回真
索引节点处理	__mark_inode_dirty	使索引节点“脏”	void __mark_inode_dirty (struct inode * inode, int flags)	inode为要标记的索引节点，flags为标志，应当为 I_DIRTY_SYNC	这是一个内部函数，调用者应当调用 mark_inode_dirty 或 mark_inode_dirty_sync
	write_inode_now	向磁盘写一个索引节点	void write_inode_ now (struct inode * inode, int sync)	inode为要写到磁盘的索引节点，sync 表示是否需要同步。	如果索引节点为脏，该函数立即把它写到给磁盘。主要由 knfsd 来使用
	clear_inode	清除一个索引节点	void clear_inode (struct inode * inode)	inode为要写清除的索引节点	由文件系统来调用该函数，告诉我们该索引节点不再有用
	invalidate_inodes	丢弃一个设备上的索引节点	int invalidate_ inodes (struct super_block * sb);	sb 为超级块	对于给定的超级块，丢弃所有的索引节点。如果丢弃失败，说明还有索引节点处于忙状态，则返回一个非 0 值。如果丢弃成功，则超级块中所有的节点都被丢弃。

续表

类别	函数名	功能	函数形成	参数	描述
索引节点处理	get_empty_inode	获得一个索引节点	struct inode * get_ empty_inode (void)	无	这个函数的调用发生在诸如网络层想获得一个无索引节点号的索引节点，或者文件系统分配一个新的、无填充信息的索引节点 成功则返回一个指向 inode 的指针，失败则返回一个 NULL 指针。返回的索引节点不在任何超级块链表中
	iunique	获得一个唯一的索引节点号	ino_t iunique (struct super_block * sb, ino_t max_reserved)	sb 为超级块，max_reserved 为最大保留索引节点号	对于给定的超级块，获得该系统上一个唯一的索引节点号。这一般用在索引节点编号不固定的文件系统中。返回的节点号大于保留的界限但是唯一。注意，如果一个文件系统有大量的索引节点，则这个函数会很慢
	insert_inode_hash	把索引节点插入到哈希表	void insert_inode_ hash (struct inode * inode)	inode为要插入的索引节点	把一个索引节点插入到索引节点的哈希表中，如果该节点没有超级块，则把它加到一个单独匿名的链中

	remove_inode_hash	从哈希表中删除一个索引节点	void remove_inode_hash (struct inode * inode)	inode 为要删除的索引节点	从超级块或匿名哈希表中删除一个索引节点
	iput	释放一个索引节点	void iput (struct inode * inode)	inode 为要释放的索引节点	如果索引节点的引用计数变为 0，则释放该索引节点，并且可以撤销它
	bmap	在一个文件中找到一个块号	int bmap (struct inode * inode, int block)	inode 为文件的索引节点，block 为要找的块	返回设备上的块号，例如，寻找索引节点 1 的块 4，则该函数将返回相对于磁盘起始位置的盘块号
	update_atime	更新访问时间	void update_atime (struct inode * inode)	inode 为要访问的索引节点	更新索引节点的访问时间，并把该节点标记为写回。这个函数自动处理只读文件系统、介质、“noatime”标志以及具有“noatime”标记者的索引节点
	make_bad_inode	由于 I/O 错误把一个索引节点标记为坏	void make_bad_inode (struct inode * inode)	inode 为要标记为坏的索引节点	由于介质或远程网络失败而造成不能读一个索引节点时，该函数把该节点标记为“坏”，并引起从这点开始的 I/O 操作失败
	is_bad_inode	是否是一个错误的 inode	int is_bad_inode (struct inode * inode)	inode 为要测试的索引节点	如果要测试的节点已标记为坏，则返回真

续表

类别	函数名	功能	函数形成	参数	描述
注册以及超级块	register_filesystem	注册一个新的文件系统	int register_filesys- tem (struct file_sys- tem_type * fs)	fs 为指向文件系统结构的指针	把参数传递过来的文件系统加到文件系统的链表中。成功则返回 0，失败则返回一个负的错误码
	unregister_filesystem	注销一个文件系统	int unregister_files- ystem (struct file_ system_type * fs)	fs 为指向文件系统结构的指针	把曾经注册到内核中的文件系统删除。如果没有找到个文件系统，则返回一个错误码，成功则返回 0 这个函数所返回的 file_sys tem_type 结构被释放或重用
	get_super	获得一个设备的超级块	struct super_block * get_super (kdev_t dev)	dev 为要获得超级块的设备	扫描超级块链表，查找在给定设备上安装的文件系统的超级块。如果没有找到，则返回空

6. Linux 的连网

套	函 数 名	功 能	函数形成	参 数	描 述

接字缓冲区函数	skb_queue_empty	检查队列是否为空	int skb_queue_empty (struct sk_buff_head * list)	list 为队列头	如果队列为空返回真，否则返回假
	skb_get	引用缓冲区	struct sk_buff * skb_get (struct sk_buff * skb)	skb 为要引用的缓冲区	对套接字缓冲区再引用一次，返回指向缓冲区的指针
	kfree_skb	释放一个 sk_buff	void kfree_skb (struct sk_buff * skb)	sk 为要释放的缓冲区	删除对一个缓冲区的引用，如果其引用计数变为 0，则释放它
	skb_cloned	缓冲区是否是克隆的	int skb_cloned (struct sk_buff * skb)	skb 为要检查的缓冲区	如果以 skb_clone 标志来产生缓冲区，并且是缓冲区多个共享拷贝中的一个，则返回真。克隆的缓冲区具有共享数据，因此在正常情况下不必对其进行写
	skb_shared	缓冲区是否是共享的	int skb_shared (struct sk_buff * skb)	skb 为要检查的缓冲区	如果有多于一个的人引用这个缓冲区就返回真
	skb_share_check	检查缓冲区是否共享的，如果是就克隆它	struct sk_buff * skb_share_check (struct sk_buff * skb, int pri)	skb 为要检查的缓冲区，pri 为内存分配的优先级	如果缓冲区是共享的，就克隆这个缓冲区，并把原来缓冲区的引用计数减 1，返回新克隆的缓冲区。如果不是共享的，则返回原来的缓冲区。当从中断状态或全局锁调用该函数时，pri 必须是 GFP_ATOMIC 内存分配失败则返回 NULL

续表

	函数名	功能	函数形成	参数	描述
套接字缓冲区函数	skb_queue_len	获得队列的长度	__u32 skb_queue_len (struct sk_buff_ head * list_)	list_ 为测量的链表	返回&sk_buff 队列的指针
	__skb_queue_head	在链表首部对一个缓冲区排队	void __skb_queue_head (struct sk_buff _head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表首部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中
	skb_queue_head	在链表首部对一个缓冲区排队	void skb_queue_head (struct sk_buff _head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表首部对一个缓冲区进行排队。这个函数此可以安全地使用。一个缓冲区不能同时放在两个链表中

	__skb_queue_tail	在链表尾部对一个缓冲区排队	void __skb_queue_tail (struct sk_buff_ head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表尾部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中
	skb_queue_tail	在链表尾部对一个缓冲区排队	void skb_queue_tail (struct sk_buff_ head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表尾部对一个缓冲区进行排队。这个函数有锁，因此可以安全地使用。一个缓冲区不能同时放在两个链表中
	__skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * __skb_dequeue (struct sk_buff_ head * list)	list 为要操作的队列	删除链表首部。这个函数不持有任何锁，因此使用时应当持有适当的锁。如果队链表为空则返回 NULL，成功则返回首部元素
	skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * skb_dequeue (struct sk_buff_head * list)	list 为要操作的队列	删除链表首部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素。
	skb_insert	插入一个缓冲区	void skb_insert (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中
	skb_append	追加一个缓冲区	void skb_append (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中
	skb_unlink	从链表删除一个缓冲区	void **skb_unlink** (struct sk_buff * *skb*);	Skb 为要删除的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作

续表

套接字缓冲区	函数名	功能	函数形成	参数	描述
	skb_dequeue_tail	从队头删除	struct sk_buff * **skb_dequeue_tail** (struct sk_buff_ head * *list*)	List 为要操作的链表	删除链表尾部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素

区函数	skb_put	把数据加到缓冲区	unsigned char * **skb_put** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充缓冲区所使用的数据区。如果扩充后超过缓冲区总长度，内核会产生警告。函数返回的指针指向所扩充数据的第 1 字节
	skb_push	把数据加到缓冲区的开始	unsigned char * **skb_push** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充在缓冲区的开始处缓冲区所使用的数据区。如果扩充后超过缓冲区首部空间的总长度，内核会产生警告。函数返回的指针指向所扩充数据的第一个字节
	skb_pull	从缓冲区的开始删除数据	unsigned char * **skb_pull** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要删除的数据长度	这个函数从链表开始处删除数据，把腾出的内存归还给首部空间。把指向下一个缓冲区的指针返回
	skb_headroom	缓冲区首部空闲空间的字节数	int **skb_headroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 首部空闲空间的字节数
	skb_tailroom	缓冲区尾部的空闲字节数	int **skb_tailroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 尾部空闲空间的字节数
	skb_reserve	调整头部的空间	void **skb_reserve** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要改变的缓冲区，len 为要删除的字节数	通过减少尾部空间，增加一个空&sk_buff 的首部空间。这仅仅适用于空缓冲区
	skb_trim	从缓冲区删除尾部	void **skb_trim** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要改变的缓冲区，len 为新的长度	通过从尾部删除数据，剪切缓冲区的长度。如果缓冲区已经处于指定的长度，则不用改变
	skb_orphan	使一个缓冲区成为孤儿	void **skb_orphan** (struct sk_buff * *skb*);	skb 是要成为孤儿的缓冲区	如果一个缓冲区当前有一个拥有者，我们就调用拥有者的析构函数，使 skb 没有拥有者。该缓冲区继续存在，但以前的拥有者不再对其“负责”
	skb_queue_purge	使一个链表空	void **skb_queue_purge** (struct sk_buff_head * *list*)	list 为要腾空的链表	删除在&sk_buff 链表上的所有缓冲区。这个函数持有链表锁，并且是原子的

续表

套	函数名	功能	函数形成	参数	描述

接字缓冲区函数	dev_alloc_skb	为发送分配一个skbuff	struct sk_buff * **dev_alloc_skb** (unsigned int *length*)	length 为要分配的长度	分配一个新的&sk_buff，并赋予它一个引用计数。这个缓冲区有未确定的头空间。用户应该分配自己需要的头空间。 如果没有空闲内存，则返回NULL。尽管这个函数是分配内存，但也可以从中断来调用
	skb_cow	当需要时拷贝 skb 的首部	struct sk_buff * **skb_cow** (struct sk_buff * *skb*, unsigned int *headroom*)	skb 为要拷贝的缓冲区，headr-oom 为需要的头空间	如果传递过来的缓冲区缺乏足够的头空间或是克隆的，则该缓冲区被拷贝，并且附加的头空间变为可用。如果没有空闲的内存，则返回空。如果缓冲区拷贝成功，则返回新的缓冲区，否则返回已存在的缓冲区
	skb_over_panic	私有函数	void **skb_over_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb 为缓冲区，sz 为大小，here 为地址	用户不可调用
	skb_under_panic	私有函数	void **skb_under_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb 为缓冲区，sz 为大小，here 为地址	用户不可调用
	alloc_skb	分配一个网络缓冲区	struct sk_buff * **alloc_skb** (unsig-ned int *size*, int *gfp_mask*)	size 为要分配的大小，gfp_mask 为分配掩码	分配一个新的&sk_buff。返回的缓冲区没有 size 大小的头空间和尾空间。新缓冲区的引用计数为 1。返回值为一缓冲区，如果失败则返回空。从中断分配缓冲区，掩码只能使用 GFP_ATOMIC 的 gfp_mask
	__kfree_skb	私有函数	void **__kfree_skb** (struct sk_buff * *skb*)	skb 为缓冲区	释放一个 sk_buff。释放与该缓冲区相关的所有事情，清除状态。这是一个内部使用的函数，用户应当调用 kfree_skb
	skb_clone	复制一个sk_buff	struct sk_buff * **skb_clone** (struct sk_buff * *skb*, int *gfp_mask*)	skb 为要克隆的缓冲区，gfp_mask 为分配掩码	复制一个&sk_buff。新缓冲区不是由套接字拥有。两个拷贝共享相同的数据包而不是结构。新缓冲区的引用计数为 1。如果分配失败，函数返回NULL，否则返回新的缓冲区。如果从中断调用这个函数，掩码只能使用 GFP_ATOMIC 的gfp_mask

续表

	函数名	功能	函数形成	参数	描述
套接字缓冲区函数	skb_copy_expand	拷贝并扩展 sk_buff	struct sk_buff * **skb_copy_expand** (const struct sk_buff * *skb*, int *newheadroom*, int *newtailroom*, int *gfp_mask*);	skb 为要拷贝的缓冲区，newhead- room 为头部的新空闲字节数，newtailroom 为尾部的新空闲字节数	既拷贝&sk_buff 也拷贝其数据，同时分配额外的空间。当调用者希望修改数据并需要对私有数据进行改变，以及给新的域更多的空间时调用该函数。失败返回 NULL，成功返回指向缓冲区的指针 返回的缓冲区其引用计数为 1。如果从中断调用，则必须传递的优先级为 GFP_ATO- MIC

7．网络设备支持

	函数名	功能	函数形成	参数	描述
驱动程序的支持	init_etherdev	注册以太网设备	truct net_device * **init_etherdev** (struct net_device * *dev*, int *sizeof_priv*)	dev 为要填充的以太网设备结构，或者要分配一个新的结构时为 NULL，sizeof_priv 是为这个以太网设备要分配的额外私有结构的大小	用以太网的通用值填充这个结构的域。如果传递过来的 dev 为 NULL，则构造一个新的结构，包括大小为 sizeof_priv 的私有数据区。强制将这个私有数据区在 32 字节（不是位）上对齐
	dev_add_pack	增加数据包处理程序	void **dev_add_pack** (struct packet_type * *pt*)	pt 为数据包类型	把一个协议处理程序加到网络栈，把参数传递来的 &packet_type 链接到内核链表中
	dev_remove_pack	删除数据包处理程序	void **dev_remove_pack** (struct packet_type * *pt*)	pt 为数据包类型	删除由 dev_add_pack 曾加到内核的协议处理程序。把 &packet_type 从内核链表中删除，一旦该函数返回，这个结构还能再用
	__dev_get_by_name	根据名字找设备	struct net_device * **__dev_get_by_name** (const char * *name*);	name 为要查找的名字	根据名字找到一个接口。必须在 RTNL 信号量或 dev_base_lock 锁的支持下调用。如果找到这个名字，则返回指向设备的指针，如果没有找到，则返回 NULL。引用计数器并没有增加，因此调用者必须小心地持有锁

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	dev_get	测试设备是否存在	int **dev_get** (const char * *name*)	name 为要测试的名字	测试名字是否存在。如果找到则返回真。为了确保在测试期间名字不被分配或删除，调用者必须持有 rtnl 信号量。这个函数主要用来与原来的驱动程序保持兼容
	__dev_get_by_index	根据索引找设备	struct net_device * **__dev_get_by_ index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。该设备的引用计数没有增加，因此调用者必须小心地关注加锁，调用者必须持有 RTNL 信号量或 dev_base _lock 锁
	dev_get_by_index	根据名字找设备	struct net_device * **dev_get_by_ index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。所返回设备的引用计数加 1，因此，在用户调用 dev_put 释放设备之前，返回指针是安全的
	dev_alloc_name	为设备分配一个名字	int **dev_alloc_ name** (struct net_device * *dev*, const char * *name*)	dev 为设备，name 为格式化字符串。	传递过来一个格式化字符串，例如 ltd，该函数试图找到一个合适的 id。设备较多时这是很低效的。调用者必须在分配名字和增加设备时持有 dev_base 或 rtnl 锁，以避免重复。返回所分配的单元号或出错返回一个复数
	dev_alloc	分配一个网络设备和名字	struct net_device * **dev_alloc** (const char * *name*, int * *err*)	name 为格式化字符串，err 为指向错误的指针	传递过来一个格式化字符串，例如 ltd，函数给该名字分配一个网络设备和空间。如果没有可用内存，则返回 NULL。如果分配成功，则名字被分配，指向设备的指针被返回。如果名字分配失败，则返回 NULL，错误的原因放在 err 指向的变量中返回。调用者必须在做这一切时持有 dev_base 或 RTNL 锁，以避免重复分配名字
	netdev_state_change	设备改变状态	void **netdev_state _change** (struct net_device * *dev*)	name 为引起通告的设备	当一个设备状态改变时调用该函数

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	dev_open	为使用而准备一个接口	int **dev_open** (str- uct net_device * *dev*)	device 为要打开的设备	以从低层到上层的过程获得一个设备。设备的私有打开函数被调用，然后多点传送链表被装入，最后设备被移到上层，并把 NETDEV_UP 信号发送给网络设备的 notifier chain 在一个活动的接口调用该函数只能是个空操作。失败则返回一个负的错误代码
	dev_close	关闭一个接口	int **dev_close** (struct net_device * *dev*)	dev 为要关闭的设备	这个函数把活动的设备移到关闭状态。向网络设备的 notifier chain 发送一个 NETDEV_GOING_DOWN。然后把设备变为不活动状态，并最终向 notifier chain 发 NETDEV_DOWN 信号
	register_netdevice_notifier	注册一个网络通告程序块	int **register_netdevice_notifier** (struct notifier_ block * *nb*)	nb 为通告程序	当网络设备的事件发生时，注册一个要调用的通告程序。作为参数传递来的通告程序被连接到内核结构，在其被注销前不能重新使用它。失败则返回一个负的错误码
	unregister_netdevice_notifier	注销一个网络通告块	int **unregister_netdevice_notifier** (struct notifier _block * *nb*)	nb 为通告程序	取消由 register_netdevice _notifer 曾注册的一个通告程序。把这个通告程序从内核结构中解除，然后还可以重新使用它。失败则返回一个负的错误码
	dev_queue_xmit	传送一个缓冲区	int **dev_queue_xmit** (struct sk_ buff * *skb*)	skb 为要传送的缓冲区	为了把缓冲区传送到一个网络设备，对缓冲区进行排队。调用者必须在调用这个函数前设置设备和优先级，并建立缓冲区。该函数也可以从中断中调用。失败返回一个负的错误码。成功并不保证帧被传送，因为也可能由于拥塞或流量调整而撤销这个帧
	netif_rx	把缓冲区传递到网络协议层	void **netif_rx** (struct sk_buff * *skb*)	skb 为要传送的缓冲区	这个函数从设备驱动程序接收一个数据包，并为上层协议的处理对其进行排队。该函数总能执行成功。在处理期间，可能因为拥塞控制而取消这个缓冲区。

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	register_gifconf	注册一个 SIOCGIF 处理程序	int **register_gifconf** (unsigned int *family*, gifconf_func_t * *gifconf*)	family 为地址族,gifconf 为处理程序	注册由地址转储例程决定的协议。当另一个处理程序替代了由参数传递过来的处理程序时，才能释放或重用后者
	netdev_set_master	建立主／从对	int **netdev_set_master** (struct net_device * *slave*, struct net_device * *master*)	slave 为从设备，master 为主设备。	改变从设备的主设备。传递 NULL 以中断连接。调用者必须持有 RTNL 信号量。失败返回一个负错误码。成功则调整引用计数，RTM_NEWLINK 发送给路由套接字，并且返回 0
	dev_set_allmulti	更新设备上多个计数	void **dev_set_all-multi** (struct net_device * *dev*, int *inc*)	dev 为设备，inc 为修改者	把接收的所有多点传送帧增加到设备或从设备删除。当设备上的引用计数依然大于 0 时，接口保持着对所有接口的监听。一旦引用计数变为 0，设备回转到正常的过滤操作。负的 inc 值用来在释放所有多点传送需要的某个资源时减少其引用计数
	dev_ioctl	网络设备的 ioctl	int **dev_ioctl** (unsigned int *cmd*, void * *arg*)	cmd 为要发出的命令,arg 为用户空间指向 ifreq 结构的指针	向设备发布 ioctl 函数。这通常由用户空间的系统调用接口调用，但有时也用作其他目的。返回值为一个正数，则表示从系统调用返回，为负数，则表示出错
	dev_new_index	分配一个索引	int **dev_new_index** (*void*)	无	为新的设备号返回一个合适而唯一的值。调用者必须持有 rtnl 信号量以确保它返回唯一的值。
	netdev_finish_unregister	完成注册	int **netdev_finish_unregister** (struct net_device * *dev*)	dev 为设备	撤销或释放一个僵死的设备。成功返回 0
	unregister_netdevice	从内核删除设备	int **unregister_netdevice** (struct net_device * *dev*)	dev 为设备。	这个函数关闭设备接口并将其从内核表删除。成功返回 0，失败则返回一个负数。

8390网卡	ei_open	打开／初始化网板	int **ei_open** (struct net_device * *dev*)	dev 为要初始化的网络设备	尽管很多注册的设备在每次启动时仅仅需要设置一次，但这个函数在每次打开设备时还彻底重新设置每件事。
	ei_close	关闭网络设备	int **ei_close** (struct net_device * *dev*)	dev 为要关闭的网络设备	ei_open 的相反操作，在仅仅在完成“ifconfig<devname> down”时使用

续表

	函数名	功能	函数形成	参数	描述
8390网卡	ethdev_init	初始化 8390 设备结构的其余部分	int **ethdev_init** (struct net_device * *dev*)	dev 为要初始化的网络设备结构	初始化 8390 设备结构的其余部分。不要用__init ()，因为这也由基于 8390 的模块驱动程序使用
	NS8390_init	初始化 8390 硬件	void **NS8390_init** (struct net_device * *dev*, int *startp*)	dev 为要初始化的设备，startp 为布尔值，非 0 启动芯片处理。	必须持以锁才能调用该函数

8．模块支持

	函数名	功能	函数形成	参数	描述
模块装入	request_module	试图装入一个内核模块	int **request_module** (const char * *module_name*)	module_name 为模块名	使用用户态模块装入程序装入一个模块。成功返回 0，失败返回一个负数。注意，一个成功的装入并不意味着这个模块在自己出错时就能卸载和退出。调用者必须检查他们所提出的请求是可用的，而不是盲目地调用。 如果自动装入模块的功能被启用，那么这个函数就不起作用。
	call_usermodehelper	启动一个用户态的应用程序	int call_usermodehelper (char * path, char ** argv, char ** envp);	path 为应用程序的路径名，argv 为以空字符结束的参数列表，envp 为以空字符结束的环境列表	运行用户空间的一个应用程序。该应用程序被异步启动。它作为 keventd 的子进程来运行，并具有 root 的全部权能。Keventd 在退出时默默地获得子进程 必须从进程的上下文中调用该函数，成功返回 0，失败返回一个负数。

内部模块支持	inter_module_register	注册一组新的内部模块数据	void inter_module_register (const char * im_name, struct module * owner, const void * userdata)	im_name 为确定数据的任意字符串，必须唯一，owner 为正在注册数据的模块，通常用 THIS_MODULE ，userdata 指向要注册的任意用户数据	检查 im_name 还没有被注册，如果已注册就发出“抱怨”。对新数据，则把它追加到 inter_module_entry 链表。

续表

	函数名	功能	函数形成	参数	描述
内部模块支持	inter_module_get	从另一模块返回任意的用户数据	const void * inter_module_get (const char * im_name)	im_name 为确定数据的任意字符串，必须唯一	如果 im_name 还没有注册，则返回 NULL。增加模块拥有者的引用计数，如果失败则返回 NULL，否则返回用户数据
	inter_module_get_request	内部模块自动调用 request_module	const void * inter_module_get_request (const char * im_name, const char * modname)	im__name 为确定数据的任意字符串，必须唯一；modname 为期望注册 m_name 的模块	如果 inter_module_get 失败，调用 request_module,然后重试
	inter_module_put	释放来自另一个模块的数据	void inter_module_put (const char * im_name)	im__name 为确定数据的任意字符串，必须唯一	如果 im_name 还没有被注册，则“抱怨”，否则减少模块拥有者的引用计数

9．硬件接口

	函 数 名	功 能	函数形成	参 数	描 述
硬件处理	Disable_irq_nosync	不用等待使一个 irq 无效	void inline **disable_irq_nosync** (unsigned int *i*rq)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效。与 disable_ irq 不同，这个函数并不确保 IRQ 处理程序的现有实例在退出前已经完成。可以从 IRQ 的上下文中调用该函数。

	Disable_irq	等待完成使一个 irq 无效	void **disable_irq** (unsigned int *irq*)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效 这个函数要等待任何挂起的处理程序在退出之前已经完成。如果你在使用这个函数，同时还持有 IRQ 处理程序可能需要的一个资源，那么，你就可能死锁。要小心地从 IRQ 的上下文中调用这个函数
	Enable_irq	启用 irq 的	void **enable_irq** (unsigned int *irq*)	irq 为中断号	重新启用这条 IRQ 线上的中断处理。在 IRQ 的上下文中调用这个函数
	Probe_irq_mask	扫描中断线的位图	unsigned int **probe_irq_mask**(unsigned long *val*)	val 为要考虑的中断掩码	扫描 ISA 总线的中断线，并返回活跃中断的位图。然后把中断探测的逻辑状态返回给它以前的值

续表

	函数名	功能	函数形成	参数	描述
MTRR 处理	Mtrr_del	删除一个内存区类型	int **mtrr_del** (int *reg*, unsigned long *base*, unsigned lon- g *size*);	reg 为由 mtrr_add 返回的寄存器，base 为物理基地址，size 为内存区大小	如果提供了寄存器 reg，则 base 和 size 都可忽略。这就是驱动程序如何调用寄存器。如果引用计数降到 0，则释放该寄存器，该内存区退回到缺省状态。成功则返回寄存器，失败则返回一个负数
PCI 支持库	pci_find_slot	从一个给定的 PCI 插槽定位 PCI	struct pci_dev * **pci_find_slot** (unsigned int *bus*, unsigned int *devfn*)	bus 为所找 PCI 设备所驻留的 PCI 总线的成员，devfn 为 PCI 插槽的成员	给定一个 PCI 总线和插槽号，所找的 PCI 设备位于 PCI 设备的系统全局链表中。如果设备被找到，则返回一个指向它的数据结构，否则返回空

PCI支持库	函数名	功能	函数形成	参数	描述
	pci_find_device	根据 PCI 标识号开始或继续搜索一个设备	struct pci_dev * **pci_find_device** (unsigned int *vendor*, unsigned int *device*, const struct pci_dev * *from*)	vendor 为要匹配的 PCI 商家 id，或要与所有商家 id 匹配的 PCI_ANY _ID，device 为要匹配的 PCI 设备 id，或要与所有商家 id 匹配的 PCI_ ANY_ ID，from 为以前搜索中找到的 PCI 设备，或对于一个新的搜索来说为空	循环搜索已知 PCI 设备的链表。如果找到与 vendor 和 device 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索
	pci_find_class	根据类别开始或继续搜索一个设备	struct pci_dev * **pci_find_class** (unsigned int *class*, const struct pci_dev * *from*)	class：根据类别名称搜索 PCI 设备 Previous：在搜索 着找到的 PCI 设备，对于新的搜索则为 NULL	循环搜索已知 PCI 设备的链表。如果找到与 class 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索
	pci_find_capability	查询设备的权能	int pci_find_capability (struct pci_dev * dev, int cap)	dev 为要查询的 PCI 设备，cap 为权能取值	断定一个设备是否支持给定 PCI 权能。返回在设备 PCI 配置空间内所请求权能结构的地址，如果设备不支持这种权能，则返回 0

续表

	函数名	功能	函数形成	参数	描述
PCI支持库	pci_set_power_state	设置一个设备电源管理的状态	int **pci_set_power_state** (struct pci_dev * *dev*, int *new_state*)	dev 为 PCI 设备，new_state 为新的电源管理声明(0 == D0, 3 == D3 等)	设置设备的电源管理状态。对于从状态 D3 的转换，并不像想象的那么简单，因为很多设备在唤醒期间忘了它们的配置空间。返回原先的电源状态
	pci_save_state	保存设备在挂起之前PCI的配置空间	int pci_save_state (struct pci_dev * dev, u32 * buffer)	dev 为我们正在处理的 PCI 设备，buffer 为持有配置空间的上下文	缓冲区必须足够大，以保持整个 PCI 2.2 的配置空间(>= 64 bytes)。

	pci_restore_state	恢复 PCI 设备保存的状态	int pci_restore_state (struct pci_dev * dev, u32 * buffer)	dev 为我们正在处理的 PCI 设备，buffer 为保存的配置空间	
	pci_enable_device	驱动程序使用设备前进行初始化	int pci_enable_device (struct pci_dev * dev)	dev 为要初始化的 PCI 设备	驱动程序使用设备前对设备进行初始化。请求低级代码启用 I/O 和内存。如果设备被挂起，则唤醒它。小心，这个函数可能失败
	pci_disable_device	使用 PCI 设备之后使其无效	void pci_disable_device (struct pci_dev * dev)	dev 为使无效的 PCI 设备	向系统发送信号，以表明系统不再使用 PCI 设备。这仅仅包括使 PCI 总线控制（如果激活）无效
	pci_enable_wake	当设备被挂起时启用设备产生 PME#	int pci_enable_wake (struct pci_dev * dev, u32 state, int enable)	dev 为对其实施操作的 PCI 设备，state 为设备的当前状态，enable 为启用或禁用"产生"的标志	当系统被挂起时，在设备的 PM 能力中设置位以产生 PME#。如果设备没有 PM 能力，则返回-EIO。如果设备支持它，则返回-EINVAL，但不能产生唤醒事件。如果操作成功，则返回 0
	pci_release_regions	释放保留的 PCI I/O 和内存资源	void pci_release_regions (struct pci_dev * pdev)	pdev 为 PCI 设备，其资源以前曾由 pci_request_regions 保留。	释放所有的 PCI I/O 和以前对 pci_request_regions 成功调用而使用的内存。只有在 PCI 区的所有使用都停止后才调用这个函数
	pci_request_regions	保留 PCI I/O 和内存资源	int pci_request_regions (struct pci_dev * pdev, char * res_ name)	pdev 为 PCI 设备，它的资源要被保留，res_name 为与资源相关的名字	把所有与 PCI 设备 pdev 相关联的 PCI 区进行标记，设备 pdev 是由属主 res_name 保留的。除非这次调用成功返回，否则不要访问 PCI 内的任何地址 成功返回 0,出错返回 EBUSY，失败时也打印警告信息

续表

	函数名	功能	函数形成	参数	描述
PCI 支持库	pci_unregister_driver	注销一个 PCI 设备	void pci_unregister_driver (struct pci_driver * drv)	drv 为要注销的驱动程序结构	从已注册的 PCI 驱动程序链表中删除驱动程序结构，对每个驱动程序所驱动的设备，通过调用驱动程序的删除函数，给它一个清理的机会，把这些设备标记为无驱动程序的

	pci_insert_device	插入一个热插拔设备	void pci_insert_device (struct pci_dev * dev, struct pci_bus * bus)	dev 为要插入的设备，bus 为 PCI 总线，设备就插入到该总线	把一个新设备插入到设备列表，并向用户空间(/sbin/hotplug）发出通知
	pci_remove_device	删除一个热插拔设备	void pci_remove_device (struct pci_dev * dev)	dev 为要删除的设备	把一个新设备从设备列表删除，并向用户空间(/sbin/hotplug）发出通知。
	pci_dev_driver	获得一个设备的 pci_driver	struct pci_driver * pci_dev_driver (const struct pci_dev * dev)	dev 为要查询的设备	返回合适的 pci_driver 结构，如果一个设备没有注册的驱动程序，则返回 NULL
	pci_set_master	为设备 dev 启用总线控制	void pci_set_master (struct pci_dev * dev)	dev 为要启用的设备	启用设备上的总线控制，并调用 pcibios_set_master 对特定的体系结构进行设置
	pci_setup_device	填充一个设备的类和映射信息	int pci_setup_device (struct pci_dev * dev)	dev 为要填充的设备结构	用有关设备的商家、类型、内存及 IO 空间地址，IRQ 线等初始化设备结构。在 PCI 子系统初始化时调用该函数。成功返回 0，设备类型未知返回-1

10．块设备

函数名	功能	函数形式	参数	描述	其他
blk_cleanup_queue	当不再需要一个请求队列时，释放一个 request_queue_t	void blk_cleanup_queue (request_queue_t * q);	q 为要释放的请求队列	blk_cleanup_queue 与 blk_init_ queue 是成对出现的。应该在释放请求队列时调用该函数；典型的情况是块设备正被注销时调用。该函数目前的主要任务是释放分配到队列中所有的 struct request 结构	低级驱动程序有希望首先完成任何重要的请求。

续表

函数名	功能	函数形式	参数	描述	其他

blk_queue_headactive	指明请求队列的头是否可以是活跃的	void blk_queue_headactive (request_queue_t * q, int active)	q 为这次申请的队列，active 为一个标志，表示队列头在哪儿是活跃的	块设备驱动程序可以选定把当前活动请求留在请求队列，只有在请求完成时才移走它。队列处理例程为安全起见把这种情况假定为缺省值，并在请求被撤销时，将不再在合并或重新组织请求时包括请求队列的头 如果驱动程序在处理请求之前从队列移走请求，它就可以在合并和重新安排中包含队列头。这可以通过以 active 标志为 0 来调用 blk_queue_headactive 如果一个驱动程序一次处理多个请求，它必须从请求队列移走他们（或至少一个） 当一个队列被插入，则假定该队列头为不活跃的	
blk_queue_make_request	为设备定义一个交替的 make_request 函数	void blk_queue_make_request (request_queue_t * q, make_request_fn * mfn)	q 为受影响设备的请求队列，mfn 为交替函数	把 buffer_heads 结构传递到设备驱动程序的常用方式为让驱动程序把请求收集到请求队列，然后让驱动程序准备就绪时把请求从那个队列移走。这种方式对很多块设备驱动程序很有效。但是，有些块设备（如虚拟设备 md 或 lvm）并不是这样，而是把请求直接传递给驱动程序，这可以通过调用 blk_queue_make_request（）函数来达到	按以上方式操作的驱动程序必须能够恰当地处理在“高内存”的缓冲区，这是通过调用 bh_kmap 获得一个内核映射，或通过调用 create_bounce 在常规内存创建一个缓冲区
blk_init_queue	为块设备的使用准备一个请求队列	void blk_init_queue (request_queue_t * q, request_fn_proc * rfn)	q 为要初始化的请求队列，rfn 为处理请求所调用的函数	如果一个块设备希望使用标准的请求处理例程，就调用该函数。当请求队列上有待处理的请求时，调用 rfn 函数	blk_init_queue 的反操作函数为 blk_cleanup_queue，当撤销块设备时调用后者（例如在模块卸载时）

续表

函数名	功能	函数形式	参数	描述	其他
generic_make_request	形成块设备的 I/O 请求	void generic_make_request (int rw, struct buffer_head * bh)	rw为I/O操作的类型，即READ、WRITE或READA，bh是内存和磁盘上的缓冲区首部	READ和WRITE的含义很明确，READA为预读。该函数不返回任何状态。请求的成功与失败，以及操作的完成是由bh->b_end_io递送的	
submit_bh	类似于上一个函数	void submit_bh (int rw, struct buffer_head * bh)	rw为I/O操作的类型，即READ、WRITE或READA，bh为描述I/O的buffer_head	该函数与与generic_make_request的目的非常类似，但submit_bh做更多的事情。	
ll_rw_block	对块设备的低级访问	void ll_rw_block (int rw, int nr, struct buffer_head * * bhs)	rw为READ、WRITE或READA，nr为数组中buffer_heads的个数，bhs为指向buffer_heads的数组	对普通文件的读／写和对块设备的读／写，都是通过调用该函数完成的	所有的缓冲区必须是针对同一设备的

11．USB 设备

函数名	功能	函数形成	参数	描述
usb_register	注册一个USB设备	Int usb_register (struct usb_driver * new_driver)	new_driver为驱动程序的USB操作	注册一个具有USB核心的USB驱动程序。只要增加一个新的驱动程序，就要扫描一系列独立的接口，并允许把新的驱动程序与任何可识别的设备相关联，成功则返回0，失败则返回一个负数
usb_scan_devices	扫描所有未申明的USB接口	Usb_scan_devices (void)	无	扫描所有未申明的USB接口，并通过“probe”函数向它们提供所有已注册的USB驱动程序。这个函数将在usb_register()调用后自动地被调用
usb_deregister	注销一个USB驱动程序	Usb_deregister (struct usb_driver * driver)	Driver为要注销的驱动程序的USB操作	从USB内部的驱动程序链表中取消指定的驱动程序
usb_alloc_bus	创建一个新的USB宿主控制器结构	Struct usb_bus * usb_alloc_bus (struct	op为指向struct usb_operations的指针，这是	创建一个USB宿主控制器总线结构，并初始化所有必要的内部对象（仅仅由USB宿主控制器使用）。如果没有可用内存，则返回NULL

		usb_operations * op)	一个总线结构	

续表

函数名	功能	函数形成	参数	描述
usb_free_bus	释放由总线结构所使用的内存	Void usb_free_bus (struct usb_bus * bus)	无	(仅仅由 USB 宿主控制器驱动程序使用)
usb_register_bus	注册具有 usb 核心的 USB 宿主控制器	Void usb_register_bus (struct usb_bus * bus);	bus 指向要注册的总线	仅仅由 USB 宿主控制器驱动程序使用

附录B 在线文档

1　Linux 源代码的获取

网站为 http://www.kernel.org/ ，在这里可以找到各种源代码版本及补丁。

2　Linux 源代码超文本交叉检索工具

国外网站：http://lxr.linux.no/ ，国内镜像网站为：http://www2.linuxforum.net/lxr/http/source

3　Linux 内核文档项目（LDP）

站点为：http://www.linuxdoc.org，该主页中还包括了有用的链接、指南、FAQ 及 HOWTO。

4　GCC 对 C 语言的扩展

站点为：http://developer.apple.com/techpubs/macosx/DeveloperTools/Compiler/Compiler.1d.html，该主页描述了标准 C 中所没有而 GCC 对 C 的扩展功能。

5　Linux 的汇编

站点为：http://www.tldp.org/HOWTO/Assembly-HOWTO。

6　Linux 开发论坛

新闻组为：comp.os.linux.development.system。专门讨论 Linux 内核的开发问题。

7　Linux 内核邮件列表

邮件列表为：linux-kernel@vger.rutger.edu。这份邮件列表的内容非常丰富，可以从中找到 Linux 内核当前开发版的最新内容。

8　Linux 的内存管理

网站为：http://linux-mm.org/，该主页描述有关内存管理的各种信息。

9　Linux 虚拟文件系统

网站为：http://www.coda.cs.cmu.edu/doc/talks/linuxvfs/，其中对 Linux 的虚拟文件系统进行了描述。

10　Linux 内核文档与源码分析

中文网站：http://www2.linuxforum.net/ker_plan/index/main.htm，这是国内 Linux 内核爱好者的论坛。

11　Linux 内核可装入模块编程

在 www.linuxdoc.org/LDP/lkmpg/mpg.html 上是 Linux 内核模块编程指南的在线文档，适合于模块编程的初学者。

另一网站 http://blacksun.box.sk/lkm.html ，是给黑客及系统管理员的的权威文档。

可以说，Linux 的在线文档数以万计，在此仅列举了与 Linux 内核相关的主要网站。

参 考 文 献

陈莉君．Linux 操作系统内核分析．人民邮电出版社，2000.3

陈莉君等译．深入理解 Linux 内核．中国电力出版社，2001.10

毛德操．胡希明．Linux 内核源代码情景分析．浙江大学出版社，2001.9

田云等．保护模式下 80386 及其编程．清华大学出版社，1993.12

艾德才等．80486／80386．系统原理与接口大全．清华大学出版社，1995.8

王鹏等译．操作系统设计与实现.电子工业出版社，1998.8

李善平等．Linux 操作系统实验教程．机械工业出版社，1999.10

ALESSANDRO RUBINI 著，LISOlEG 等译．Linux 设备驱动程序．中国电力出版社，2000.4